

# Java 编程规范 (第三版)

## The Java Language Specification

(Third Edition)

James Gosling  
Bill Joy  
[美] Guy Steele 著  
Gilad Bracha  
陈宗斌 沈金河 译



**Java 之父的全新力作**  
**Java 语言的标准参考书**



中国电力出版社  
www.infopower.com.cn

国外经典程序设计系列

# Java 编程规范

## The Java Language Specification

( Third Edition )

James Gosling  
[ 美 ] Bill Joy 著  
Guy Steele  
Gilad Bracha  
陈宗斌 沈金河 译



中国电力出版社

[www.infopower.com.cn](http://www.infopower.com.cn)



Java Language Specification, 3<sup>rd</sup> Edition (ISBN 0-321-24678-0)

James Gosling, Bill Joy, Guy Steele, Gilad Bracha

Copyright ©1996-2005 Sun Microsystems, Inc.

Original English Language Edition Published by Addison-Wesley.

All rights reserved.

Translation edition published by PEARSON EDUCATION ASIA LTD and CHINA ELECTRIC POWER PRESS, Copyright © 2006.

本书翻译版由 Pearson Education 授权中国电力出版社独家出版、发行。

未经出版者书面许可, 不得以任何方式复制或抄袭本书的任何部分。

本书封面贴有 Pearson Education 防伪标签, 无标签者不得销售。

北京市版权局著作权合同登记号 图字: 01-2006-3388 号

### 图书在版编目 (CIP) 数据

Java 编程规范 (第三版) / (美) 高斯林 (Gosling, J.) 等编著; 陈宗斌, 沈金河译.

—北京: 中国电力出版社, 2006

(国际经典程序设计系列)

书名原文: Java Language Specification, 3<sup>rd</sup> Edition

ISBN 7-5083-4292-5

I.J... II.①高...②陈...③沈... III.JAVA 语言—程序设计 IV.TP312

中国版本图书馆 CIP 数据核字 (2006) 第 060159 号

丛 书 名: 国际经典程序设计系列

书 名: Java 编程规范 (第三版)

编 著: (美) James Gosling, Bill Joy, Guy Steele, Gilad Bracha

翻 译: 陈宗斌 沈金河

责任编辑: 牛贵华

出版发行: 中国电力出版社

地址: 北京市三里河路 6 号 邮政编码: 100044

电话: (010) 88515918 传 真: (010) 88518169

印 刷: 北京市铁成印刷厂

开本尺寸: 185×233 印 张: 29.25 字 数: 665 千字

书 号: ISBN 7-5083-4292-5

版 次: 2006 年 7 月北京第 1 版 2006 年 7 月第 1 次印刷

定 价: 49.80 元

版权所有 翻印必究

“当我说话时，”矮胖子轻蔑地说，“我爱那个字是什么意思，它便是什么意思，丝毫不差。”

爱丽丝抗议道：“问题是，你能随心所欲地为一个词下定义吗？”

矮胖子回答得倒也干脆，他说：“问题是，这只是谁说了算而已。”

——Lewis Carroll, 《镜中世界》

# 目 录

前言	
第二版前言	
第三版前言	
第 1 章 简介 .....	1
1.1 示例程序 .....	4
1.2 符号 .....	4
1.3 预定义类和接口的关系 .....	4
1.4 参考文献 .....	5
第 2 章 语法 .....	7
2.1 与环境无关的语法 .....	7
2.2 词法语法 .....	7
2.3 语义语法 .....	7
2.4 语法符号 .....	8
第 3 章 词法结构 .....	11
3.1 Unicode .....	11
3.2 词法转换 .....	12
3.3 Unicode 转义符 .....	12
3.4 行终止符 .....	13
3.5 输入元素和标记 .....	14
3.6 空白 .....	15
3.7 注释 .....	15
3.8 标识符 .....	16
3.9 关键字 .....	17
3.10 字面值 .....	18
3.11 分隔符 .....	26
3.12 运算符 .....	26
第 4 章 类型、值和变量 .....	27
4.1 各种类型和值 .....	28

4.2	基本类型和值 .....	28
4.3	引用类型和值 .....	35
4.4	类型变量 .....	39
4.5	参数化类型 .....	41
4.6	类型擦除 .....	45
4.7	可具体化的类型 .....	45
4.8	原生类型 .....	46
4.9	交集类型 .....	49
4.10	子类型化 .....	49
4.11	在何处使用类型 .....	51
4.12	变量 .....	53
<b>第 5 章</b>	<b>转换和提升 .....</b>	<b>60</b>
5.1	转换的种类 .....	62
5.2	赋值转换 .....	71
5.3	方法调用转换 .....	76
5.4	字符串转换 .....	77
5.5	强制转换 .....	77
5.6	数值提升 .....	82
<b>第 6 章</b>	<b>名称 .....</b>	<b>85</b>
6.1	声明 .....	86
6.2	名称和标识符 .....	86
6.3	声明的作用域 .....	88
6.4	成员和继承 .....	92
6.5	确定名称的含义 .....	95
6.6	访问控制 .....	104
6.7	完全限定的名称和规范名称 .....	109
6.8	命名约定 .....	110
<b>第 7 章</b>	<b>包 .....</b>	<b>116</b>
7.1	包成员 .....	116
7.2	包的主机支持 .....	117
7.3	编译单元 .....	119
7.4	包声明 .....	119
7.5	导入声明 .....	121
7.6	顶级类型声明 .....	126
7.7	惟一的包名称 .....	128



第 8 章	类 .....	130
8.1	类声明 .....	131
8.2	类成员 .....	143
8.3	字段声明 .....	147
8.4	方法声明 .....	159
8.5	成员类型声明 .....	180
8.6	实例初始化语句 .....	181
8.7	静态初始化语句 .....	181
8.8	构造函数声明 .....	182
8.9	枚举 .....	189
第 9 章	接口 .....	197
9.1	接口声明 .....	197
9.2	接口成员 .....	200
9.3	字段（常量）声明 .....	201
9.4	抽象方法声明 .....	203
9.5	成员类型声明 .....	205
9.6	注释类型 .....	206
9.7	注释 .....	213
第 10 章	数组 .....	219
10.1	数组类型 .....	219
10.2	数组变量 .....	220
10.3	数组创建 .....	221
10.4	数组访问 .....	221
10.5	数组：一个简单的示例 .....	221
10.6	数组初始化语句 .....	222
10.7	数组成员 .....	223
10.8	数组的 Class 对象 .....	224
10.9	字符的数组不是一个 String .....	224
10.10	数组存储异常 .....	224
第 11 章	异常 .....	226
11.1	异常的起因 .....	227
11.2	异常的编译时检查 .....	227
11.3	异常处理 .....	229
11.4	异常的示例 .....	231
11.5	异常层次结构 .....	232

<b>第 12 章 执行 .....</b>	<b>234</b>
12.1 虚拟机启动 .....	234
12.2 加载类和接口 .....	236
12.3 链接类和接口 .....	237
12.4 初始化类和接口 .....	239
12.5 创建新的类实例 .....	243
12.6 类实例的终结 .....	246
12.7 卸载类和接口 .....	249
12.8 程序退出 .....	250
<b>第 13 章 二进制兼容性 .....</b>	<b>251</b>
13.1 二进制的形式 .....	252
13.2 二进制兼容性是什么，不是什么 .....	255
13.3 包的演变 .....	255
13.4 类的演变 .....	256
13.5 接口的演变 .....	269
<b>第 14 章 块和语句 .....</b>	<b>271</b>
14.1 语句的正常结束和突然结束 .....	271
14.2 块 .....	272
14.3 本地类声明 .....	273
14.4 局部变量声明语句 .....	274
14.5 语句 .....	278
14.6 空语句 .....	279
14.7 标签语句 .....	280
14.8 表达式语句 .....	280
14.9 if 语句 .....	281
14.10 assert 语句 .....	282
14.11 switch 语句 .....	285
14.12 while 语句 .....	288
14.13 do 语句 .....	289
14.14 for 语句 .....	291
14.15 break 语句 .....	294
14.16 continue 语句 .....	296
14.17 return 语句 .....	297
14.18 throw 语句 .....	298
14.19 synchronized 语句 .....	299
14.20 try 语句 .....	300

14.21	不可到达语句 .....	305
<b>第 15 章</b>	<b>表达式 .....</b>	<b>309</b>
15.1	计算、表示和结果 .....	309
15.2	变量作为值 .....	310
15.3	表达式的类型 .....	310
15.4	精确浮点数表达式 .....	310
15.5	表达式和运行时检查 .....	311
15.6	计算的正常和突然结束 .....	312
15.7	求值顺序 .....	313
15.8	主表达式 .....	317
15.9	类实例创建表达式 .....	320
15.10	数组创建表达式 .....	325
15.11	字段访问表达式 .....	329
15.12	内存调用表达式 .....	332
15.13	数组访问表达式 .....	362
15.14	后缀表达式 .....	365
15.15	一元运算符 .....	366
15.16	强制转换表达式 .....	369
15.17	乘法运算符 .....	370
15.18	加运算符 .....	373
15.19	移位运算符 .....	377
15.20	关系运算符 .....	378
15.21	相等运算符 .....	380
15.22	位和逻辑运算符 .....	382
15.23	条件与运算符&& .....	383
15.24	条件或运算符   .....	383
15.25	条件运算符?: .....	384
15.26	赋值运算符 .....	385
15.27	表达式 .....	396
15.28	常量表达式 .....	396
<b>第 16 章</b>	<b>明确赋值 .....</b>	<b>398</b>
16.1	明确赋值和表达式 .....	402
16.2	语明确赋值和语句 .....	406
16.3	明确赋值和参数 .....	412
16.4	明确赋值和数组初始化方法 .....	412
16.5	明确赋值和枚举常量 .....	412

16.6	明确赋值和匿名类 .....	413
16.7	明确赋值和成员类型 .....	413
16.8	明确赋值和静态初始化方法 .....	413
16.9	明确赋值、构造函数和实例初始化方法 .....	414
第 17 章	线程和锁 .....	415
17.1	锁 .....	415
17.2	示例中的符号 .....	416
17.3	不正确同步的程序出现意外行为 .....	416
17.4	内存模型 .....	418
17.5	Final 字段语义 .....	428
17.6	字分开 .....	432
17.7	double 和 long 的非原子处理 .....	433
17.8	等待集合和通知 .....	433
17.9	休眠和转交 .....	435
第 18 章	语法 .....	437
18.1	Java 编程语言的语法 .....	437



# 前 言

Java™编程语言最初被称为 Oak，由 James Gosling 设计，旨在用于嵌入式消费类电子产品应用程序上。使用几年之后，加上 Ed Frank、Patrick Naughton、Jonathan Payne 和 Chris Warth 的巨大贡献，该语言被重新定位为以 Internet 为目标，经过重新命名，从实质上加以修改，就形成了本书要讨论的 Java 语言。Java 语言的最终形式由 James Gosling、Bill Joy、Guy Steele、Richard Tuck、Frank Yellin 和 Arthur van Hoff 确定下来，在此过程中得到了 Graham Hamilton、Tim Lindholm 以及许多朋友及同事的大力帮助。

Java 编程语言是一种多用途的、并发的、基于类的、面向对象的编程语言，且特别设计为具有尽可能少的实现依赖性。它允许应用程序开发人员编写一次程序，然后可以在 Internet 上到处运行。

本书试图详细说明 Java 语言的语法和语义规范。我们试图在本书中说明每一种语言结构的行为，以便让所有实现都接受同一个程序。除了时间相关性或无法确定因素之外，如果有足够的时间和足够的内存空间，用 Java 编程语言所写的程序应该可以在所有机器和所有实现中计算出相同的结果。

我们相信 Java 编程语言是一种将被广泛使用的成熟语言。尽管如此，我们仍期望未来几年内该语言会有所进步。我们试图让它以完全与现有应用程序兼容的方式来管理这些演变。为了达到此目的，我们试图写一些相对新的语言版本。编译器和系统能够同时支持几个具有完全兼容性的版本。

Java 平台的许多研究和实验都在进行中。我们鼓励进行此项工作，并且将继续与外部组织协作开发来改进语言和平台。例如，我们已经收到几个有意义的参数化类型的提议。在技术上较难的领域中，尤其是最新的技术上，这种研究合作是很有必要的。

我们诚挚地感谢通过优秀的反馈、协助和鼓励对本书作出贡献的人。

尤其要感谢的是：Tom Cargill、Peter Deutsch、Paul Hilfinger、Masayuki Ida、David Moon、Steven Muchnick、Charles L. Perkins、Chris Van Wyk、Steve Vinoski、Philip Wadler、Daniel Weinreb 和 Kenneth Zadeck，他们完整、细心和缜密地对本书的草稿进行了审阅。我们由衷地感谢他们非常自愿的努力。

我们还要感谢 Stephen Adams、Bowen Alpern、Glenn Ammons、Leonid Arbuzov、Kim Bruce、Edwin Chan、David Chase、Pavel Curtis、Drew Dean、William Dietz、David Dill、Patrick Dussud、Ed Felten、John Giannandrea、John Gilmore、Charles Gust、Warren Harris、Lee Hasiuk、Mike Hendrickson、Mark Hill、Urs Hoelzle、Roger Hoover、Susan Flynn Hummel、

Christopher Jang、Mick Jordan、Mukesh Kacker、Peter Kessler、James Larus、Derek Lieber、Bill McKeeman、Steve Naroff、Evi Nemeth、Robert O'Callahan、Dave Papay、Craig Partridge、Scott Pfeffer、Eric Raymond、Jim Roskind、Jim Russell、William Scherlis、Edith Schonberg、Anthony Scian、Matthew Self、Janice Shepherd、Kathy Stark、Barbara Steele、Rob Strom、William Waite、Greg Weeks 和 Bob Wilson 对本书的审阅、提问、评论和建议（此人员名单是从我们的 E-mail 记录自动生成的。如果遗漏了任何人，我们对此深表歉意）。

在改进语言的定义和本书的演示文稿形式上，所有这些审阅者的反馈对我们来说都是非常宝贵的。我们感谢他们的辛勤劳动。本书中任何残留的错误——我们希望这会很少——都是我们而非他们的责任。

我们感谢 Francesca Freedman 和 Doug Kramer 对印刷和出版事务的协助。我们感谢 Adobe Systems Incorporated 的 Dan Mills 对寻找可能的字体提供的协助。

Sun 公司的许多同事在很多方面帮助了我们。本系列丛书的编辑 Lisa Friendly 协调着我们与 Addison-Wesley 的关系。Susan Stambaugh 帮助管理着给审阅者的几百份草稿副本。我们收到了 Ben Adida、Ole Agesen、Ken Arnold、Rick Cattell、Asmus Freytag、Norm Hardy、Steve Heller、David Hough、Doug Kramer、Nancy Lee、Marianne Mueller、Akira Tanaka、Greg Tarsy、David Ungar、Jim Waldo、Ann Wollrath、Geoff Wyant 和 Derek White 有价值的帮助和技术上的建议。我们感谢 Alan Baratz、David Bowen、Mike Clary、John Doerr、Jon Kannegaard、Eric Schmidt、Bob Sproull、Bert Sutherland 和 Scott McNealy 的领导和鼓励。

在研究和验证遍布本书的引文的过程中，Columbia 大学的在线 Bartleby Library（在线网址为 <http://www.cc.columbia.edu/acis/bartleby/>）对我们来说非常宝贵。下面是一个例子：

他们以其他著作的丰富经验润色他们贫乏的书籍，

——Robert Burton（1576~1640）

我们非常感激在 Project Bartleby 工作的人，节省了我们大量的工作，并且重新激起了我们对沃尔特·惠特曼作品的欣赏之情。

感谢在写这本书时我们在整理工作中所用到的工具和服务：电话、旭夜快递、台式工作站、激光打印机、复印机、排版软件、字体库、电子邮件、WWW，当然还有 Internet。我们住在三个不同的州，但是仍能不费吹灰之力与其他人以及审阅者合作。荣誉给予那些常年努力不懈的、让这些优秀的工具和服务快速且可靠工作的成千上万的人们。

感谢 Addison-Wesley 的 Mike Hendrickson、Katie Duffy、Simone Payment 和 Rosa Aimée González 在本书付梓期间所给予的帮助、鼓励和耐心。我们还要感谢这些文字编辑。

感谢 Rosemary Simpson 在非常繁忙的日常工作中为本书制作了索引。

最后，我们非常感谢我们的家人和朋友在最近疯狂工作的一年里给予了我们爱和支持。

Brian Kernighan 和 Dennis Ritchie 在他们的《The C Programming Language》中写道，他们觉得 C 语言“随着经验的增长越来越好用”。如果你喜欢 C，我们想你也将会喜欢 Java 编程

语言。我们希望它对你也一样好用。

James Gosling

加利福尼亚州，库珀蒂诺

Bill Joy

科罗拉多州，阿斯彭

Guy Steele

马萨诸塞州，切姆斯福德

1996 年 7 月

## 第二版前言

---

……金字塔已经岿然屹立一千年；而生物则要么进化，要么毁灭。  
——Alan Perlis, 《Structure and Interpretation of Computer Programs》一书的前言

过去几年里，Java™编程语言获得了空前的成功。这种成功带来了挑战：随着用户的增长，对于语言和库的需求也出现了爆炸性的增长。为了应对这个挑战，语言得到了发展（幸运的是，不是爆炸性的），库也是如此。

本书第二版反映出了这些发展。它整合了 Java 编程语言自 1996 年本书第一版出版后所发生的改变。1997 年，这些大量的改变被写入了 Java 平台的 1.1 版本中，并考虑增加嵌入的类型声明。稍后的修改是浮点操作。另外，这一版本收编了包含方法搜索和二进制兼容性的重要说明和修正。

本书定义了目前现有的语言。Java 编程语言仍会继续发展。在写这本书时，就有一些正在进行的计划，通过 Java Community Process 利用泛型类型和断言扩展语言、优化内存模型，等等。但是，如果等到这些工作都有了结果之后再出版本书的第二版，肯定是不合适的。

现在 API 库的规范已经太大了，不适合纳入本书的范围，而且它们还在不断发展。因此，API 规范已从本书中删除。库规范可以在 <http://java.sun.com> 网站上找到；该规范目前仅仅专注于 Java 编程语言。

读者可以将对本书的意见发送至邮箱：[jls@java.sun.com](mailto:jls@java.sun.com)。要学习最新的 Java 2 平台或是下载最新的 Java 2 SDK 版本，请访问 <http://java.sun.com>。关于 Java 系列图书的更新信息，包括本书的勘误以及即将出版的书籍，请访问：<http://java.sun.com/Series>。

许多人对本书直接或间接地做出了贡献。Tim Lindholm 作为技术编辑贡献非常大，尤其是在浮点数问题上。本书如果没有他的话，将会缺少色彩。Lisa Friendly 作为本系列丛书的编辑，提供了很多的鼓励和建议，对此我非常感谢。

David Bowen 是第一个建议我关心 Java 平台规范的人。我很感谢他引导我进入这个丰富的领域。

感谢 Java 编程语言中的嵌套类型之父 John Rose，在我试图详细准确地解释嵌套类型时，他给予了我无限的亲切和支持。

很多人对第二版提供了很有价值的意见。特别感谢 Ergnosis 的 Roly Perera 以及 Leonid



Arbouzov 和他在 Novosibirsk 的 Sun Java 平台一致性小组的同事: Konstantin Bobrovsky、Natalia Golovleva、Vladimir Ivanov、Alexei Kaigorodov、Serguei Katkov、Dmitri Khukhro、Eugene Latkin、Ilya Neverov、Pavel Ozhdikhin、Igor Pyankov、Viatcheslav Rybalov、Serguei Samoilidi、Maxim Sokolnikov 和 Vitaly Tchaiko。他们完整地阅读了早期的草稿,对大大提高本书的准确性有很大的帮助。

非常感激 Martin Odersky、Andrew Bennett 以及 Sun javac 编译器小组过去和现在的成员: Iris Garcia、Bill Maddox、David Stoutamire 和 Todd Turnidge。他们非常努力地确保参考实现符合本规范。还有许多有意义的技术交流,我感谢他们和我在 Sun 公司的其他同事: Lars Bak、Joshua Bloch、Cliff Click、Robert Field、Mohammad Gharahgouzloo、Ben Gomes、Steffen Grarup、Robert Griesemer、Graham Hamilton、Gordon Hirsch、Peter Kessler、Sheng Liang、James McIlree、Philip Milne、Srdjan Mitrovic、Anand Palaniswamy、Mike Paleczny、Mark Reinhold、Kenneth Russell、Rene Schmidt、David Ungar、Chris Vick 和 Hong Zhang。我的经理 Tricia Jordan 是耐心、体量和理解下属的典范。还要感谢 Java 2 标准版的主管 Larry Abrahams 对这个工作的支持。

下列这些人都提供了对本书很有用的意见: Godmar Bak、Hans Boehm、Philippe Charles、David Chase、Joe Darcy、Jim des Rivieres、Sophia Drossopoulou、Susan Eisenbach、Paul Haahr、Urs Hoelzle、Bart Jacobs、Kent Johnson、Mark Lillibridge、Norbert Lindenberg、Phillipe Mulet、Kelly O'Hair、Bill Pugh、Cameron Purdy、Anthony Scian、Janice Shepherd、David Shields、John Spicer、Lee Worall 和 David Wragg。

Suzette Pelouch 和 Doug Kramer、Atul Dambalkar 一起,利用他们的 FrameMaker 专业知识,为本书索引提供了非常宝贵的帮助。Addison-Wesley 的 Mike Hendrickson 和 Julie Dinicola 相当友好,他们对本书的出版给予了无限的帮助。

就我个人来说,我要感谢我的妻子 Weihong 对我的爱和支持。

最后,衷心感谢我的合作者 James Gosling、Bill Joy 和 Guy Steele 邀请我参加这项工作。这是件相当愉快且荣耀的事情。

Gilad Bracha  
加利福尼亚州, 洛斯阿尔托斯  
2000 年 4 月

这是为女性准备的辞典。  
男性辞典大致相同,但不完全相同。注意,这一段是最为关键的不同。  
谨慎选择吧!

——Milorad Pavic,《哈扎尔辞典》女性版

## 第三版前言

本书的这个版本代表该语言的历史中一组最大的变革。该语言近来把泛型、注释、断言、自动装箱、拆箱、枚举类型、foreach 循环、可变元数方法和静态导入等技术都添加到了其中。除了断言之外，其他所有的技术都是在 2004 年秋季的 5.0 版本上新增的。

本书的第三版反映了这些发展。它集成了自 2000 年第二版发布以来对 Java 编程语言所做的所有变革。

该语言在过去 4 年里获得了长足的发展。不幸的是，浓缩一种商业上成功的编程语言的精华是不现实的——而只会使它增长得越来越大。在兼容性约束下和广泛的使用和用户冲突要求下，管理这种增长是一个很大的挑战。我只能希望借助本规范成功地满足这项挑战：时间会告诉我们一切。

读者可以把对本规范的评论发送至：[jls@java.sun.com](mailto:jls@java.sun.com)。要了解关于 Java 平台的最新信息，或者下载最新的 J2SE 版本，请访问 <http://java.sun.com>。可以在 <http://java.sun.com/Series> 上找到关于 Java 系列（包括本书第三版的勘误表以及即将出版的书籍）的更新信息。

本规范建立于许多人的努力之上，包括 Sun 公司内部和外部的人员。

最至关重要的贡献是由那些把规范变成真实软件的人做出的。这些人中首屈一指的是 javac 的维护者，javac 是 Java 编程语言的参考编译器。

Neal Gafter 是在集成和产品化这里描述的重大变革这个重要时期的“javac 先生”。Neal 的贡献和生产力可以被公正地描述为如史诗般的宏伟。如果没有他，我们简直不能完成这项任务。此外，他的见解和技能对设计所有的新语言特性做出了巨大的贡献。对该语言的这个版本来说，没有哪个人应该得到比他更多的赞誉——但是，对于其缺点的任何指责都应该落到我本人以及许多 JSR 专家组的成员身上！

Neal 着手研究了新的挑战，Peter von der Ahé 接替和继承了他的工作，他继续改进和加强了实现。在 Neal 参与进来之前，在前一版本完成时，Bill Maddox 负责 javac，并且他通过他们早期的工作发掘了诸如泛型和断言之类的特性。

另一个值得单独提及的人是 Joshua Bloch。Josh 参与了无休止的语言设计讨论，在多个专家组担任主席，并且为 Java 平台做出了关键的贡献。公平地讲，Josh 和 Neal 对本书的关心超过了我本人！

本规范的许多部分都是由多个不同的专家组在 Java 社区组织的框架中开发的。

最普及的语言变革集合是 JSR-014 的成果：添加泛型到 Java 编程语言中。JSR-04 专家

组的成员有：Norman Cohen、Christian Kemper、Martin Odersky、Kresten Krab Thorup、Philip Wadler 和我本人。在早期阶段，成员还包括 Sven-Eric Panitz 和 Steve Marx。所有这些都值得对他们的参与表示谢意。

JSR-014 代表一项空前的努力，在非常严格的兼容性要求之下，从根本上扩展广泛使用的编程语言的类型系统。经过一段漫长且费力的设计和实现过程之后，我们到达了当前的语言扩展。远在 JSR 发起泛型研究以前，Martin Odersky 和 Philip Wadler 就已经创建了一种称为 Pizza 的实验语言，用于探索涉及的思想。1998 年春天，David Stoutamire 和我本人基于这些思想开始与 Martin 和 Phil 合作，这就产生了 GJ。当 JSR-014 专家组把各个成员召集在一起时，就把 GJ 选作扩展 Java 编程语言的基础。Martin Odersky 实现了 GJ 编译器，并且他的实现变成了 javac 的基础（javac 开始于 JDK 1.3，尽管直到 JDK 1.5 才启用泛型）。

泛型类型系统的核心的理论基础在很大程度上归功于 Martin Odersky 和 Phil Wadler 的专业知识。后来，利用通配符扩展了该系统。这些基于 Atsushi Igarashi 和 Mirko Viroli 的工作，它本身构建于 Kresten Thorup 和 Mads Torgersen 的早期工作之上。通配符最初是作为 Sun 和 Aarhus 大学之间合作的一部分而设计和实现的。Neal Gafter 和我本人代表 Sun，连同 Erik Ernst 和 Mads Torgersen 以及代表 Aarhus 的 Peter von der Ahé 和 Christian Plesner-Hansen 一起参与了这次合作。感谢 Ole Lehrmann-Madsen 鼓励和支持那项工作。

Joe Darcy 和 Ken Russell 实现了对泛型反射的大部分特定支持。Neal Gafter、Josh Bloch 和 Mark Reinhold 做了大量的工作来泛型化 JDK 库。

有些人必须得到表扬，他们对泛型设计的评论导致了显著的差别。Alan Jeffrey 通过指出原始类型系统中的细微缺陷，从而对 JSR-14 做出了至关重要的贡献。Bob Deen 建议为低有界通配符使用“? super T”语法。

JSR-201 包括一系列变革：自动装箱、枚举、foreach 循环、可变元数方法和静态导入。JSR-201 专家组的成员包括：Cédric Beust、David Biesack、Joshua Bloch（联合主席）、Corky Cartwright、Jim des Rivieres、David Flanagan、Christian Kemper、Doug Lea、Changshin Lee、Tim Peierls、Michel Trudeau 和我本人（联合主席）。枚举和 foreach 循环主要是由 Josh Bloch 和 Neal Gafter 设计的。如果没有 Neal 专门的设计工作，可变元数方法将永远不会加入到该语言中（这里没有提及实现它们的一些小问题）。

Josh Bloch 自己勇敢地承担了 JSR-175 的职责，即把注释添加到该语言中。JSR-175 专家组的成员包括：Cédric Beust、Joshua Bloch（主席）、Ted Farrell、Mike French、Gregor Kiczales、Doug Lea、Deeptendu Majunder、Simon Nash、Ted Neward、Roly Perera、Manfred Schneider、Blake Stone 和 Josh Street。Neal Gafter 作为主要的贡献者，也照例工作在最前沿。

本版本的另一个变革是完全修订了 Java 内存模型，它是由 JSR-133 承担的。JSR-133 专家组的成员包括：Hans Boehm、Doug Lea、Tim Lindholm（联合主席）、Bill Pugh（联合主席）、Martin Trotter 和 Jerry Schwarz。内存模型的主要技术作家是 Sarita Adve、Jeremy Manson 和 Bill Pugh。本书中关于 Java 内存模型的章节事实上几乎全都是他们的工作，只是做了少许编辑上的修订。Joseph Bowbeer、David Holmes、Victor Luchangco 和 Jan-Willem

Maessen 也做出了重大贡献。第 12 章中论及终结 (finalization) 的关键几节内容在很大程度上也应归功于这项工作，特别是 Doug Lea 的工作。

许多人对本版本提供了有价值的评论。

我希望向 Archibald Putt 表达我的谢意，他对本书提供了见解和鼓励。他的著作总是鼓舞人心。再一次感谢 Joe Darcy，他引荐了我们，提出了许多有用的意见，并且在数值问题和十六进制值的设计方面做出了他的独特贡献。

Sun 的许多同事（过去的和现在的）提供了有用的反馈和讨论，并且以无数种方式帮助了本书的出版，他们是：Andrew Bennett、Martin Buchholz、Jerry Driscoll、Robert Field、Jonathan Gibbons、Graham Hamilton、Mimi Hills、Jim Holmlund、Janet Koenig、Jeff Norton、Scott Seligman、Wei Tao 和 David Ungar。

特别感谢我的经理 Laurie Tolson，感谢她在我编写这些规范的漫长过程中所提供的支持。

对本规范提供了有价值意见的还有：Scott Annanian、Martin Bravenboer、Bruce Chapman、Lawrence Gonsalves、Tim Hanson、David Holmes、Angelika Langer、Pat Lavarre、Phillipe Mulet 和 Cal Varnson。

Addison-Wesley 的 Ann Sellers、Greg Doench 和 John Fuller 极其耐心地保证了本书的问世，而不计较本书多次错过了截稿日期。

与以往一样，我感谢我的妻子 Weihong 和我的儿子 Teva 的支持和协作。

Gilad Bracha  
加利福尼亚州，洛斯阿尔托斯  
2005 年 1 月



# 第 1 章

## 简 介

如果我看得更远些，那是因为我站在巨人的肩膀上。  
——牛顿

Java 编程语言是一种通用、并发、基于类且面向对象的语言。它非常简单，足以让许多程序员可以灵活自如地驾驭该语言。Java 编程语言与 C 和 C++ 相关，但是在组织方式上有较大的差别，Java 具有 C 和 C++ 遗漏的许多方面，同时兼有其他语言包括的一些思想。它旨在成为一种生产语言，而不是一种研究语言，并且正是如此，就像 C. A. R. Hoare 在其关于语言设计的经典论文中所建议的一样，这种设计避免了包括新的、未经测试的特性。

Java 编程语言是强类型化的。本规范清楚区分了能够并且必须在编译时检测到的编译时错误和那些在运行时发生的错误。编译时通常包括将程序转换成机器无关的字节码表示。运行时活动包括加载和链接执行程序所需要的类，生成可选的机器代码，动态优化程序和执行实际的程序。

Java 编程语言是一种相对高级的语言，这是由于机器表示的细节无法通过语言得到。它包括自动存储管理（通常使用垃圾收集器），以避免显式存储单元分配（如 C 的 free 或 C++ 的 delete）的安全问题。高性能垃圾收集的实现可以有限地中止对系统编程和实时应用的支持。该语言不包括任何非安全构造，如无索引检查的数组访问，这是由于这种非安全构造会导致程序以未指定的方式工作。

Java 编程语言通常编译成《The Java™ Virtual Machine Specification, Second Edition》（Addison-Wesley, 1999 年）中定义的字节码指令集和二进制格式。

本书的结构组织如下：

第 2 章描述了用于介绍 Java 语言的词汇及语句语法的语法和符号。

第 3 章描述了 Java 编程语言的词法结构，它基于 C 和 C++。Java 语言是用 Unicode 字符集编写的。它支持在只支持 ASCII 的系统上编写 Unicode 字符。

第 4 章描述了类型、值和变量。类型被细分成基本类型和引用类型。

基本类型被定义成在所有机器上和所有实现中都一样，并且具有多种不同的大小，包括：2 的补码整数、单精度和双精度 IEEE 754 标准浮点数、boolean 型、Unicode 字符 char 型。基本类型的值不共享状态。

引用类型包括：类类型、接口类型和数组类型。引用类型是由动态创建的对象（类或数组的实例）实现的。可以有多个引用指向一个对象。所有对象（包括数组）支持类 `Object` 的方法，`Object` 类是类层次结构的（惟一）根部。预定义的 `String` 类支持 `Unicode` 字符串。有用于把基本值包装在对象内部的类。在许多情况下，包装和解包装是由编译器自动执行的（在这种情况下，包装称为装箱，解包装称为拆箱）。类和接口声明可能是泛型的，也就是说，它们可以通过其他引用类型进行参数化。之后就可以通过特定类型的参数调用这种声明。

变量是类型化的存储位置。基本类型的变量存储的正好是那种基本类型的值。类类型的变量可以存储空引用，或者指向一个对象的引用，该对象的类型是那个类类型或者那个类类型的任何子类。接口类型的变量可以存储空引用，或者指向实现接口的任何类的实例的引用。数组类型的变量可以存储空引用，或者指向一个数组的引用。`Object` 类类型的变量可以存储空引用，或者指向任何对象（类实例或数组）的引用。

第5章描述了转换和数值提升。转换可以更改编译时类型，有时还会更改表达式的值。这些转换包括基本类型和引用类型之间的装箱和拆箱转换。数值提升用于把数值运算符的操作数转换成可以在其中执行操作的公共类型。Java 语言中没有什么漏洞：在运行时会检查引用类型上的强制类型转换，以确保类型安全。

第6章描述了声明和名称，以及如何确定名称的含义（意义）。在使用类型及其成员前，Java 语言不需要声明它们。声明的顺序仅对于局部变量、局部类以及类或接口中字段的初始值设定项的顺序是重要的。

Java 编程语言提供了对名称作用域的控制，并且支持对包、类和接口的成员的外部访问的限制。这有助于通过把类型的实现在其用户和那些扩展它的人之间区分开，来编写大型程序。这里描述了会使程序更可读的推荐命名约定。

第7章描述了程序的结构，它被组织进类似于 `Modula` 模块的包中。包的成员有类、接口和子包。包被细分成编译单元。编译单元包含类型声明，并且可以从其他包导入类型以为其提供短名称。包具有分层的命名空间中的名称，并且 `Internet` 域名系统通常可用于形成独特的包名称。

第8章描述了类。类的成员有类、接口、字段（变量）和方法。类的变量依赖于类而存在。类的方法无需引用特定的对象即可操作。实例变量是在对象（类的实例）中动态创建的。实例方法则是在类的实例上调用的；这种实例在方法的执行期间成为当前对象 `this`，从而支持面向对象的编程风格。

类支持单实现继承，其中每个类的实现是由单一超类的实现派生的，并且最终由类 `Object` 派生。类类型的变量可以引用那个类或那个类的任何子类的实例，从而允许现有的方法通过多态方式使用新的类型。

类通过 `synchronized` 方法支持并发编程。这些方法声明了在其执行期间可能发生的受查异常（checked exception），它允许进行编译时检查，以确保异常情况会得到处理。对象可以声明 `finalize` 方法，在对象被丢弃前，将通过垃圾收集器调用该方法，从而允许对象清理它们的状态。

为了简单起见，Java 语言既没有独立于类的实现的“头部”声明，也没有独立的类型

和类层次结构。

一种特殊的类形式 `enums` (枚举) 支持以类型安全的方式定义较小的值集合及其操作。与其他语言中的枚举 (`enumeration`) 不同的是, `enums` 是对象, 并且可以有其自己的方法。

第 9 章描述了接口类型, 它声明一组抽象方法、成员类型和常量。不相关的类可以实现相同的接口类型。接口类型的变量可以包含指向实现接口的任何对象的引用。Java 语言支持多重接口继承。

注释类型是特殊化的接口, 用于对声明作注释。这种注释不允许以任何方式影响 Java 编程语言中的程序的语义。但是, 它们给多种不同的工具提供了有用的输入。

第 10 章描述了数组。数组访问包括边界检查。数组是动态创建的对象, 并且可以赋予 `Object` 类型的变量。与多维数组相比, Java 语言对数组的支持性更好。

第 11 章描述了异常, 它是不可恢复执行的, 并且与 Java 语言的语义和并发机制完全集成。异常有三类: 受查异常 (`checked exception`)、运行时异常 (`run-time exception`) 和错误。编译器确保受查异常会得到正确处理, 这是通过仅当方法或构造函数声明异常时, 要求方法或构造函数可以产生受查异常来实现的。这提供了在编译时检查异常处理程序的存在情况, 并且帮助在大型程序中进行编程。大多数用户定义的异常应该都是受查异常。Java 虚拟机检测到的程序中的无效操作会导致运行时异常, 如 `NullPointerException`。错误来源于虚拟机检测到的失败, 如 `OutOfMemoryError`。大多数简单的程序不会尝试处理错误。

第 12 章描述了程序执行期间发生的活动。程序通常存储为表示编译过的类和接口的二进制文件。这些二进制文件可以加载进 Java 虚拟机中, 链接到其他的类和接口, 并进行初始化。

在初始化后, 可以使用类的方法和类的变量。可以实例化某些类以创建该类类型的对象。对象是类的实例, 也包含该类的每个超类的实例, 并且对象创建涉及到递归创建这些超类的实例。

当某个对象不再被引用时, 就可以用垃圾收集器回收它。如果对象声明了一个终结器 (`finalizer`), 在对象被回收前, 该终结函数会给对象最后一次机会来清理那些不会被释放的资源。当不再需要某个类时, 可以卸载它。

第 13 章描述了二进制兼容性, 详细说明了如果更改类型, 则会对那些使用更改的类型但未重新编译的其他类型所产生的影响。对于那些在连续的版本系列中通常通过 Internet 广泛分布的类型, 本章介绍的内容正是这些类型的开发人员所感兴趣的事项。无论何时更改了某个类型, 良好的程序开发环境会自动重新编译相关的代码, 因此大多数程序员不必考虑这些细节。

第 14 章描述了块和语句, 它们是基于 C 和 C++ 的。Java 语言没有 `goto` 语句, 但是包含有带标签的 `break` 和 `continue` 语句。与 C 不同的是, Java 编程语言在控制流程语句中需要 `boolean` (或 `Boolean`) 表达式, 并且不会把类型隐式转换成 `boolean` 类型 (除非通过拆箱), 这是希望在编译时捕获更多的错误。`synchronized` 语句提供了基本的对象级监视锁定。`try` 语句可以包括 `catch` 和 `finally` 子句, 以防止非局部控制转移。

第 15 章描述了表达式。为了提高决断能力和可移植性, 本文档详尽说明了表达式求值的 (明显) 顺序。对于重载的方法和构造函数, 在编译时通过从那些适用的方法或构造函数

数中选择最特定的方法或构造函数来对其进行解析。

第 16 章描述了 Java 语言确保局部变量在使用前被明确设置的精确方式。虽然所有其他的变量都会被自动初始化为一个默认值，但是 Java 编程语言不会自动初始化局部变量，以避免掩盖编程错误。

第 17 章描述了线程和锁的语义，它们以基于监视的并发机制为基础，这种机制最初是由 Mesa 编程语言引入的。Java 编程语言为支持高性能实现的共享内存型多处理器指定了一种内存模型。

第 18 章介绍了 Java 语言的语义语法。

## 1.1 示例程序

本书中给出的大多数示例程序都是可执行的，并且具有如下类似形式：

```
class Test {  
    public static void main(String[] args) {  
        for (int i = 0; i < args.length; i++)  
            System.out.print(i == 0 ? args[i] : " " + args[i]);  
        System.out.println();  
    }  
}
```

在使用 Sun 的 Java 2 Platform Standard Edition Development Kit 软件的 Sun 工作站上，这个类存储在文件 `Test.java` 中，可以通过给出以下命令来编译和执行它：

```
javac Test.java  
java Test Hello, world.  
产生输出：  
Hello, world.
```

## 1.2 符号

在整本书中，我们都引用了从 Java 和 Java 2 平台中提取出来的类和接口。无论何时我们使用单独一个标识符  $N$  来引用未在本书的示例中定义类或接口，则预期的引用是 `java.lang` 包中名为  $N$  的类或接口。我们为来自于除 `java.lang` 以外的包的类或接口使用规范名称（6.7 节）。

我们在本书中引用《The Java™ Virtual Machine Specification》时都指的是第二版，它是经 JSR 924 修订过的。

## 1.3 预定义类和接口的关系

如前所述，本规范通常引用 Java 和 Java 2 平台的类。特别地，有些类具有与 Java 编



程语言的特殊关系。示例包括诸如 `Object`、`Class`、`ClassLoader`、`String`、`Thread` 这样的类，以及 `java.lang.reflect` 包及其他包中的类和接口。Java 语言的定义约束了这些类和接口的行为，但是本文档并没有提供它们的完整规范。读者可以参考 Java 平台规范的其他部分，以了解这些详细的 API 规范。

因此，本文档并没有描述反射（reflection）的细节。许多语言构造与反射 API 具有相似之处，但是这里一般不作讨论。因此，例如，当我们列出可用于创建对象的方式时，我们一般不会包括反射 API 也可以完成的方式。读者应该知道这些额外的机制，即使它们在本书中并未提到。

## 1.4 参考文献

- Apple Computer. *Dylan™ Reference Manual*. Apple Computer Inc., Cupertino, California. September 29, 1995. See also <http://www.cambridge.apple.com>.
- Bobrow, Daniel G., Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. *Common Lisp Object System Specification*, X3J13 Document 88-002R, June 1988; appears as Chapter 28 of Steele, Guy. *Common Lisp: The Language*, 2nd ed. Digital Press, 1990, ISBN 1-55558-041-6, 770–864.
- Ellis, Margaret A., and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, Massachusetts, 1990, reprinted with corrections October 1992, ISBN 0-201-51459-1.
- Goldberg, Adele and Robson, David. *Smalltalk-80: The Language*. Addison-Wesley, Reading, Massachusetts, 1989, ISBN 0-201-13688-0.
- Harbison, Samuel. *Modula-3*. Prentice Hall, Englewood Cliffs, New Jersey, 1992, ISBN 0-13-596396.
- Hoare, C. A. R. *Hints on Programming Language Design*. Stanford University Computer Science Department Technical Report No. CS-73-403, December 1973. Reprinted in SIGACT/SIGPLAN Symposium on Principles of Programming Languages. Association for Computing Machinery, New York, October 1973.
- IEEE Standard for Binary Floating-Point Arithmetic. ANSI/IEEE Std. 754-1985. Available from Global Engineering Documents, 15 Inverness Way East, Englewood, Colorado 80112-5704 USA; 800-854-7179.
- Kernighan, Brian W., and Dennis M. Ritchie. *The C Programming Language*, 2nd ed. Prentice Hall, Englewood Cliffs, New Jersey, 1988, ISBN 0-13-110362-8.
- Madsen, Ole Lehrmann, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the Beta Programming Language*. Addison-Wesley, Reading, Massachusetts, 1993, ISBN 0-201-62430-3.
- Mitchell, James G., William Maybury, and Richard Sweet. *The Mesa Programming Language*, Version 5.0. Xerox PARC, Palo Alto, California, CSL 79-3, April 1979.

Stroustrup, Bjarne. *The C++ Programming Language*, 2nd ed. Addison-Wesley, Reading, Massachusetts, 1991, reprinted with corrections January 1994, ISBN 0-201-53992-6.

Unicode Consortium, The. *The Unicode Standard: Worldwide Character Encoding*, Version 1.0, Volume 1, ISBN 0-201-56788-1, and Volume 2, ISBN 0-201-60845-6. Updates and additions necessary to bring the Unicode Standard up to version 1.1 may be found at <http://www.unicode.org>.

Unicode Consortium, The. *The Unicode Standard, Version 2.0*, ISBN 0-201-48345-9. Updates and additions necessary to bring the Unicode Standard up to version 2.1 may be found at <http://www.unicode.org>.

Unicode Consortium, The. *The Unicode Standard, Version 4.0*, ISBN 0-321-18578-1. Updates and additions may be found at <http://www.unicode.org>.



语法，甚至知道如何控制一个国王。  
——莫里哀，《可笑的女学究》（1672）第二幕第六场

本章描述了本规范中使用的与环境无关的语法，这些语法用于定义程序的词法和语义结构。

### 2.1 与环境无关的语法

与环境无关的语法包括许多产生式（production）。每个产生式都有一个作为其“左部”的称为非终结符（nonterminal）的抽象符号和一系列的一个或多个非终结符，以及一个作为其“右部”的称为终结符（terminal）的符号。对于每种语法，终结符取自于指定的字母表。

给定的与环境无关的语法开始于一个句子，该句子包括单独一个突出的非终结符，称为目标符（goal symbol），这种语法指定了一种语言，即可能的终结符序列的集合，它们可能来自于用产生式（其左部是非终结符）的右部反复替换序列中的任何非终结符。

### 2.2 词法语法

第3章中给出了Java编程语言的词法语法（lexical grammar）。这种语法把Unicode字符集中的字符作为其终结符。它定义了一组开始于目标符Input（3.5节）的产生式，它们描述了Unicode字符序列（3.1节）是如何转换成一系列输入元素的（3.5节）。

这些输入元素以及丢弃的空白（3.6节）和注释（3.7节）构成了Java编程语言的语义语法的终结符，并且被称为标记（token）（3.5节）。这些标记是Java编程语言的标识符（3.8节）、关键字（3.9节）、值（literal）（3.10节）、分隔符（3.11节）和运算符（3.12节）。

### 2.3 语义语法

第4章、第6～10章、第14及15章给出了Java编程语言的语义语法（syntactic grammar）。

这种语法把词法语法定义的标记作为其终结符。它定义了一组开始于目标符 *CompilationUnit* 的产生式 (7.3节)，它们描述了标记序列如何构成语法上正确的程序。

## 2.4 语法符号

在本规范的全部内容中，当文本直接引用终结符时，都会在词法和语法的产生式中以等宽字体显示它们。它们将完全按照书写的那样出现在程序中。

非终结符显示为斜体。非终结符的定义是由非终结符的名称引入的，它是通过后接一个冒号来定义的。这样，非终结符的一个或多个可选右部将接着后续的行。例如，语义定义：

*IfThenStatement*:

**if** ( *Expression* ) *Statement*

指出非终结符 *IfThenStatement* 表示标记 **if**，其后接着左括号标记，再接着一个 *Expression* (表达式)，然后接着右括号标记，最后接着 *Statement* (语句)。

另一个示例如下，语义定义：

*ArgumentList*:

*Argument*

*ArgumentList* , *Argument*

指出 *ArgumentList* 可能表示单一的 *Argument* 或 *ArgumentList*，其后接着逗号，再接着 *Argument*。*ArgumentList* 的这种定义是递归式的，也就是说，它是依据其自身定义的。其结果是 *ArgumentList* 可能包含任何正参数。非终结符的这种递归式定义很常见。

下标 “*opt*” 可能出现在终结符或非终结符的后面，指示可选符号 (optional symbol)。包含可选符号的替代选择实际上指定了两个右部，一个省略了可选元素，另一个则包括了它。

这意味着：

*BreakStatement*:

**break** *Identifier*<sub>*opt*</sub> ;

是下列产生式的方便缩写：

*BreakStatement*:

**break** ;

**break** *Identifier* ;

以及：

*BasicForStatement*:

**for** ( *ForInit*<sub>*opt*</sub> ; *Expression*<sub>*opt*</sub> ; *ForUpdate*<sub>*opt*</sub> ) *Statement*

是下列产生式的方便缩写：

*BasicForStatement*:

**for** ( ; *Expression*<sub>*opt*</sub> ; *ForUpdate*<sub>*opt*</sub> ) *Statement*

**for** ( *ForInit* ; *Expression*<sub>opt</sub> ; *ForUpdate*<sub>opt</sub> ) *Statement*

这依次又是下列产生式的缩写：

*BasicForStatement*:

**for** ( ; ; *ForUpdate*<sub>opt</sub> ) *Statement*

**for** ( ; *Expression* ; *ForUpdate*<sub>opt</sub> ) *Statement*

**for** ( *ForInit* ; ; *ForUpdate*<sub>opt</sub> ) *Statement*

**for** ( *ForInit* ; *Expression* ; *ForUpdate*<sub>opt</sub> ) *Statement*

这依次又是下列产生式的缩写：

*BasicForStatement*:

**for** ( ; ; ) *Statement*

**for** ( ; ; *ForUpdate* ) *Statement*

**for** ( ; *Expression* ; ) *Statement*

**for** ( ; *Expression* ; *ForUpdate* ) *Statement*

**for** ( *ForInit* ; ; ) *Statement*

**for** ( *ForInit* ; ; *ForUpdate* ) *Statement*

**for** ( *ForInit* ; *Expression* ; ) *Statement*

**for** ( *ForInit* ; *Expression* ; *ForUpdate* ) *Statement*

因此，非终结符*BasicForStatement*实际上具有8个可选的右部。

对于比较长的右部，可以通过充分缩进第二行，在第二行接着编写其内容，如下：

*ConstructorDeclaration*:

*ConstructorModifiers*<sub>opt</sub> *ConstructorDeclarator*

*Throws*<sub>opt</sub> *ConstructorBody*

这定义了非终结符*ConstructorDeclaration*的一个右部。

当在语法定义中的冒号后面接着单词“one of”时，它们表示下面一行或多行上的每个终结符是一个可选的定义。例如，词法语法包含如下产生式：

*ZeroToThree*: one of

0 1 2 3

这只是下列产生式方便的缩写：

*ZeroToThree*:

0

1

2

3

当词法产生式作为一个标记出现时，它表示组成这个标记的字符序列。因此，词法语法产生式中的如下定义：

*BooleanLiteral*: one of

true false

是下列产生式的简写：

*BooleanLiteral*:

t r u e  
f a l s e

词法产生式的右部可以指定不允许使用短语“but not”进行某些扩展，从而指示要排除的扩展，如*InputCharacter*产生式（3.4节）和*Identifier*产生式（3.8节）所示：

*InputCharacter*:

*UnicodeInputCharacter* but not CR or LF

*Identifier*:

*IdentifierName* but not a *Keyword* or *BooleanLiteral* or *NullLiteral*

最后，有少数几个非终结符是通过罗马字体的描述性短语来描述的，在这种情况下，列出所有的替代选择是不切实际的：

*RawInputCharacter*:

any Unicode character

# 词法结构

词典编纂者：编写词典的人，辛苦劳作的人。  
——Samuel Johnson, 《Dictionary》(1755)

本章详细说明了 Java 编程语言的词法结构。

程序是用 Unicode (3.1 节) 编写的，但是提供了词法转换 (3.2 节)，以使 Unicode 转义符 (3.3 节) 可用于包括任何只使用 ASCII 字符的 Unicode 字符。行终止符被定义 (3.4 节) 成支持现有主机系统的不同转换，同时维持一致的行号。

源于词法转换的 Unicode 字符被简化成一系列输入元素 (3.5 节)，包括空白 (3.6 节)、注释 (3.7 节) 和标记。标记包括语义语法的标识符 (3.8 节)、关键字 (3.9 节)、值 (3.10 节)、分隔符 (3.11 节) 和运算符 (3.12 节)。

## 3.1 Unicode

程序是用 Unicode 字符集编写的。可以在以下站点上找到有关这种字符集及其关联的字符编码的信息：

<http://www.unicode.org>

Java 平台一直在跟踪 Unicode 规范的演进。关于类 Character 的文档中详细说明了公布的 Java 平台版本使用的准确 Unicode 版本。

Java 编程语言 1.1 以前的版本使用 Unicode 版本 1.1.5。在以下 Java 编程语言版本中升级了 Unicode 标准的更新版本：JDK 1.1 (升级到 Unicode 2.0)、JDK 1.1.7 (升级到 Unicode 2.1)、J2SE 1.4 (升级到 Unicode 3.0) 和 J2SE 5.0 (升级到 Unicode 4.0)。

Unicode 标准最初被设计成一种固定宽度的 16 位字符编码。后来，它被更改成允许需要多于 16 位表示的字符。合法码点 (code point) 的范围现在是 U+0000~U+10FFFF，使用十六进制的 U+n 表示法。码点大于 U+FFFF 的字符称为增补字符。为了只使用 16 位单元表示字符的完整范围，Unicode 标准定义了一种称为 UTF-16 的编码。在这种编码中，增补字符被表示成 16 位代码单元对，第一个代码单元来自于高半代理范围 (U+D800~

U+DBFF)，第二个来自于低半代理范围（U+DC00～U+DFFF）。对于范围在 U+0000～U+FFFF 之间的字符，码点的值与 UTF-16 代码单元的值是相同的。

Java 编程语言使用 UTF-16 编码来表示 16 位代码单元系列中的文本。少数 API（主要在 Character 类中）使用 32 位整数将码点表示成独立的实体。Java 平台提供了用于在两种表示之间进行转换的方法。

本书在讨论相关表示时，使用术语“码点”和“UTF-16 代码单元”；而在讨论不相关的表示时，则使用泛型（generic）术语“字符”。

除了注释（3.7 节）、标识符，以及字符和字符串的内容（3.10.4 节和 3.10.5 节）之外，程序中的所有输入元素（3.5 节）都只是用 ASCII 字符[以及产生 ASCII 字符的 Unicode 转义符（3.3 节）]构成的。ASCII（ANSI X3.4）是美国信息交换标准码。Unicode 字符编码的前 128 个字符就是 ASCII 字符。

## 3.2 词法转换

原始 Unicode 字符流将使用以下三个词法转换步骤转换成一系列标记，这三个步骤是依次进行的：

（1）把原始 Unicode 字符流中的 Unicode 转义符（3.3 节）转换成相应的 Unicode 字符。形如 \uxxxx（其中 xxxx 是十六进制值）的 Unicode 转义符表示 UTF-16 代码单元，其编码是 xxxx。这一转换步骤允许只使用 ASCII 字符来表达任何程序。

（2）将第（1）步得到的 Unicode 流转换成输入字符和行终止符的流（3.4 节）。

（3）将第（2）步得到的输入字符和行终止符的流转换成一系列输入元素（3.5 节），在丢弃空白（3.6 节）和注释（3.7 节）之后，它们包含的就是标记（3.5 节），这些标记是语义语法的终结符（2.3 节）。

在每一步中会使用尽可能长的转换，即使结果最终不会生成正确的程序，而另一种词法转换却会生成正确的程序也是如此。因此，输入字符 a--b 将被标记化（3.5 节）为 a,--,b，它不是任何语法上正确的程序的一部分，即使标记 a,-,-,b 可以是语法上正确的程序的一部分。

## 3.3 Unicode 转义符

实现首先会识别其输入中的 Unicode 转义符，将后接 4 个十六进制数字的 ASCII 字符 \u 转换成具有指定十六进制值的 UTF-16 代码单元（3.1 节），并不加更改地传递所有其他的字符。表示增补字符需要两个连续的 Unicode 转义符。这一转换步骤将会产生一系列 Unicode 输入字符：

*UnicodeInputCharacter:*

*UnicodeEscape*

*RawInputCharacter*

*UnicodeEscape:*



*\ UnicodeMarker HexDigit HexDigit HexDigit HexDigit*

*UnicodeMarker:*

*u*

*UnicodeMarker u*

*RawInputCharacter:*

*any Unicode character*

*HexDigit: one of*

*0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F*

这里的\、u和十六进制数字都是ASCII字符。

除了语法隐含的处理外，对于每个反斜杠\原始输入字符，输入处理必须考虑其前面有多少个连续的其他\字符，将其与非\字符或输入流的开始位置隔开。如果这个数是偶数，那么这个\就能够开启一个Unicode转义符；如果这个数是奇数，那么这个\就不能够开启一个Unicode转义符。例如，原始输入“\\u2297 = \u 2297”将会产生11个字符“\\u 2297 = ⊗”（\u2297是字符“⊗”的Unicode编码）。

如果合格的\后面没有接u，那么就将其视为*RawInputCharacter*，并保留转义的Unicode流部分。如果合格的\后面接一个或多个u，并且最后一个u后面没有接4个十六进制数字，那么就会出现编译时错误。

Unicode转义符产生的字符不会参与进一步的Unicode转义。例如，原始输入\u005cu005a会产生6个字符\u005a，因为005c是\的Unicode值。它不会产生字符z，z是Unicode字符005a，这是因为\u005c中的\未被解释成进一步的Unicode转义的起始字符。

Java编程语言指定了把用Unicode编写的程序转换成ASCII的标准方式，这将把程序更改成一种可以被基于ASCII的工具处理的形式。这种转换涉及到通过添加一个额外的u把程序源文本中的任何Unicode转义符转换成ASCII——例如\uxxxx变成\uuxxxx——同时把源文本中的非ASCII字符转换成Unicode转义符（每个转义符包含单独一个u）。

这种转换版本可以被Java编程语言的编译器（“Java编译器”）同等地接受，并且可以表示完全一样的程序。随后，可以通过把其中存在多个u的每个转义符序列转换成一系列少一个u的Unicode字符，同时把带有单个u的每个转义符序列转换成相应的单个Unicode字符，即可从这种ASCII形式中恢复原样的Unicode源文本。

实现应该使用\uxxxx表示法作为输出格式，以在无法使用合适的字体时显示Unicode字符。

### 3.4 行终止符

实现接下来通过识别行终止符来把Unicode输入字符序列划分进各行中。行的这个定义确定了由Java编译器或其他系统组件产生的行号。它还指定了注释终止形式//（3.7节）。

*LineTerminator:*

*the ASCII LF character, also known as “newline”*

the ASCII CR character, also known as "return"

the ASCII CR character followed by the ASCII LF character

*InputCharacter:*

*UnicodeInputCharacter* but not CR or LF

行是通过ASCII字符CR、LF或CR LF终止的。其后紧接LF的两个字符CR将被看作是一个行终止符，而不是两个。

结果是一系列行终止符和输入字符，它们是标记化过程第（3）步的终结符。

### 3.5 输入元素和标记

转义处理（3.3节）引出的输入字符和行终止符以及输入行识别（3.4节）将被简化成一系列输入元素。不属于空白（3.6节）或注释（3.7节）的输入元素都是标记。标记是语义语法（2.3节）的终结符。

下列产生式指定了这个过程：

*Input:*

*InputElements*<sub>opt</sub> *Sub*<sub>opt</sub>

*InputElements:*

*InputElement*

*InputElements InputElement*

*InputElement:*

*WhiteSpace*

*Comment*

*Token*

*Token:*

*Identifier*

*Keyword*

*Literal*

*Separator*

*Operator*

*Sub:*

the ASCII SUB character, also known as "control-Z"

空白（3.6节）和注释（3.7节）可用于分隔标记，如果多个标记是紧接在一起的，则有可能以另一种方式进行标记化。例如，对于输入中的ASCII字符“-”和“=”，仅当它们中间没有插入空白或注释时，才可以构成运算符标记“=”（3.12节）。

为了兼容某些操作系统，如果ASCII SUB字符（\u001a或control-Z）是转义的输入流中的最后一个字符，则将其忽略。

考虑结果输入流中的两个标记x和y。如果x在y前面，那么我们就称x位于y左边，或者

$y$ 位于 $x$ 右边。

例如，在下面这个简单的代码段中：

```
class Empty {
}
```

我们称`}`标记位于`{`标记的右边，即使在纸张上的这个二维表示中，`}`标记出现在`{`标记的左下方也是如此。这种关于词语“左边”和“右边”的用法约定，允许我们说二进制运算符的右操作数或者赋值的左部。

## 3.6 空白

空白被定义为ASCII空格、水平制表符、换页符以及行终止符（3.4节）。

*WhiteSpace:*

the ASCII SP character, also known as “space”  
 the ASCII HT character, also known as “horizontal tab”  
 the ASCII FF character, also known as “form feed”  
*LineTerminator*

## 3.7 注释

注释有两种类型：

`/* 文本 */`      传统注释：ASCII 字符`/*`和 ASCII 字符`*/`之间的所有文本都会被忽略（如同在 C 和 C++ 中一样）。  
`// 文本`          单行注释：从 ASCII 字符`//`到行尾的所有文本都会被忽略（如同在 C++ 中一样）。

这些注释是通过下列产生式正式指定的：

*Comment:*

*TraditionalComment*  
*EndOfLineComment*

*TraditionalComment:*

`/* CommentTail`

*EndOfLineComment:*

`// CharactersInLineopt`

*CommentTail:*

`* CommentTailStar`  
`NotStar CommentTail`

*CommentTailStar:*

`/`

*\* CommentTailStar*  
*NotStarNotSlash CommentTail*

*NotStar:*

*InputCharacter* but not *\**  
*LineTerminator*

*NotStarNotSlash:*

*InputCharacter* but not *\** or */*  
*LineTerminator*

*CharactersInLine:*

*InputCharacter*  
*CharactersInLine InputCharacter*

这些产生式隐含有以下所有性质:

- 注释不能嵌套。
- 在开始于//的注释中, /\*和\*/没有特殊的意义。
- 在开始于/\*或/\*\*的注释中, //没有特殊的意义。

因此, 下列文本:

```
/* this comment /* // /** ends here: */
```

是单独一条完整的注释。

词法语法暗示注释不会出现在字符(3.10.4节)或字符串(3.10.5节)内。

## 3.8 标识符

标识符是 Java 字母和 Java 数字的无限长序列, 必须以 Java 字母开头。标识符不能具有与关键字(3.9节)、布尔值(3.10.3节)或空值(3.10.7节)相同的拼写(Unicode 字符序列)。

*Identifier:*

*IdentifierChars* but not a *Keyword* or *BooleanLiteral* or *NullLiteral*

*IdentifierChars:*

*JavaLetter*  
*IdentifierChars JavaLetterOrDigit*

*JavaLetter:*

any Unicode character that is a Java letter (see below)

*JavaLetterOrDigit:*

any Unicode character that is a Java letter-or-digit (see below)

字母和数字可以取自整个 Unicode 字符集, 它支持当今世界中使用的大多数书写脚本, 包括针对汉语、日语和朝鲜语的大型集。这允许程序员在他们的程序中使用以他们的本地

语言书写的标识符。

“Java 字母”是方法 `Character.isJavaIdentifierStart(int)` 会为其返回 `true` 的字符。“Java 字母或数字”是方法 `Character.isJavaIdentifierPart(int)` 会为其返回 `true` 的字符。

Java 字母包括大写和小写的 ASCII 拉丁字母 A~Z (`\u0041~\u005a`) 和 a~z (`\u0061~\u007a`), 并且出于历史的原因, 还包括 ASCII 下划线 (`_` 或 `\u005f`) 和美元符号 (`$` 或 `\u0024`)。应当仅在机器生成的源代码中使用 `$` 字符, 或者仅用它来很少地访问遗留系统上预先存在的名称。

“Java 数字”包括 ASCII 数字 0~9 (`\u0030~\u0039`)。

仅当两个标识符完全一样, 即它们的每个字母或数字具有相同的 Unicode 字符时, 这两个标识符才是相同的。

具有相同外观的标识符可能仍然是不同的。例如, 对于包含以下单个字母的标识符: 拉丁文大写字母 A (`A`, `\u0041`)、拉丁文小写字母 A (`a`, `\u0061`)、希腊文大写字母阿尔法 (`Α`, `\u0391`)、西里尔小写字母 A (`а`, `\u0430`) 和数学粗斜体小写 A (`ₐ`, `\ud835\udc82`), 它们全都是不同的。

Unicode 复合字符不同于分解的字符。例如, 在排序时, 可以认为带撇号的拉丁文大写字母 A (`Á`, `\u00c1`) 与其后紧接着一个非空白撇号 (`'`, `\u0027`) 的拉丁文大写字母 A (`A`, `\u0041`) 相同, 但是它们在标识符中是不同的。参见《The Unicode Standard》(第 1 卷) 第 412 页, 以了解关于分解的详细信息, 并且参见该著作的第 626~627 页, 以了解关于排序的详细信息。标识符的示例如下:

```
String i3  απετη  MAX_VALUE  isLetterOrDigit
```

### 3.9 关键字

下列由 ASCII 字母构成的字符序列是保留的, 以用作关键字, 而不能用作标识符 (3.8 节):

*Keyword: one of*

<code>abstract</code>	<code>continue</code>	<code>for</code>	<code>new</code>	<code>switch</code>
<code>assert</code>	<code>default</code>	<code>if</code>	<code>package</code>	<code>synchronized</code>
<code>boolean</code>	<code>do</code>	<code>goto</code>	<code>private</code>	<code>this</code>
<code>break</code>	<code>double</code>	<code>implements</code>	<code>protected</code>	<code>throw</code>
<code>byte</code>	<code>else</code>	<code>import</code>	<code>public</code>	<code>throws</code>
<code>case</code>	<code>enum</code>	<code>instanceof</code>	<code>return</code>	<code>transient</code>
<code>catch</code>	<code>extends</code>	<code>int</code>	<code>short</code>	<code>try</code>
<code>char</code>	<code>final</code>	<code>interface</code>	<code>static</code>	<code>void</code>
<code>class</code>	<code>finally</code>	<code>long</code>	<code>strictfp</code>	<code>volatile</code>
<code>const</code>	<code>float</code>	<code>native</code>	<code>super</code>	<code>while</code>

关键字 `const` 和 `goto` 是保留的, 即使它们当前未使用也是如此。如果这些 C++ 关键字不正确地出现在程序中, 这可以允许 Java 编译器产生更好的错误消息。

虽然 `true` 和 `false` 看上去可以作为关键字，但是从技术上讲它们是布尔值（3.10.3 节）。同样，虽然 `null` 看上去可以作为关键字，但是从技术上讲它是空值（3.10.7 节）。

## 3.10 字面值

字面值（*literal*）是基本类型（4.2 节）、*String* 类型（4.3.3 节）或空类型（4.1 节）的值的源代码表示：

*Literal:*

*IntegerLiteral*  
*FloatingPointLiteral*  
*BooleanLiteral*  
*CharacterLiteral*  
*StringLiteral*  
*NullLiteral*

### 3.10.1 整数

参见第 4.2.1 节，以了解有关整数类型和值的一般性讨论。

整数可以表示成十进制（基数为 10）、十六进制（基数为 16）或八进制（基数为 8）：

*IntegerLiteral:*

*DecimalIntegerLiteral*  
*HexIntegerLiteral*  
*OctalIntegerLiteral*

*DecimalIntegerLiteral:*

*DecimalNumeral IntegerTypeSuffix<sub>opt</sub>*

*HexIntegerLiteral:*

*HexNumeral IntegerTypeSuffix<sub>opt</sub>*

*OctalIntegerLiteral:*

*OctalNumeral IntegerTypeSuffix<sub>opt</sub>*

*IntegerTypeSuffix: one of*

*1 L*

如果整数带有 ASCII 字母 `L` 或 `l`（`ell`）的后缀，则它是 `long` 类型；否则它就是 `int` 类型（4.2.1 节）。优先采用后缀 `L`，这是因为字母 `l`（`ell`）通常难以与数字 `1`（一）区分开。

十进制数字可以是单个 ASCII 字符 `0`，表示整数 `0`；或者包含一个 `1~9` 之间的 ASCII 数字，可以选择在其后接一个或多个 `0~9` 之间的 ASCII 数字，表示一个正整数：

*DecimalNumeral:*

*0*

*NonZeroDigit Digits<sub>opt</sub>*



*Digits:*

*Digit*

*Digits Digit*

*Digit:*

0

*NonZeroDigit*

*NonZeroDigit: one of*

1 2 3 4 5 6 7 8 9

十六进制数字包含前导ASCII字符0x, 或者在0x后面接有一个或多个ASCII十六进制数字, 可以表示正整数、0或负整数。值为10~15的十六进制数字分别用ASCII字母a~f或A~F表示; 用作十六进制数字的每个字母可以为大写或小写形式。

*HexNumeral:*

0 x *HexDigits*

0 X *HexDigits*

*HexDigits:*

*HexDigit*

*HexDigit HexDigits*

为了帮助说明, 下面重复使用了第3.3节中的下列产生式:

*HexDigit: one of*

0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F

八进制数字包含一个ASCII数字0, 后接一个或多个0~7之间的ASCII数字, 可以表示正整数、0或负整数。

*OctalNumeral:*

0 *OctalDigits*

*OctalDigits:*

*OctalDigit*

*OctalDigit OctalDigits*

*OctalDigit: one of*

0 1 2 3 4 5 6 7

注意: 八进制数字包含两个或多个数字; 0 总是被看作是十进制数字——实际上, 数字0、00 和 0x0 都表示完全相同的整数值。

int 类型最大的十进制数值是 2147483647 ( $2^{31}$ )。在可能出现 int 值的任何地方都可能出现 0~2147483647 之间的所有十进制值, 但是值 2147483648 只能作为一元“非”运算符“-”的操作数出现。

int 类型最大的正十六进制值和八进制值分别是 0x7fffffff 和 017777777777, 它们都等于 2147483647 ( $2^{31} - 1$ )。int 类型最小的负十六进制值和八进制值分别是

0x80000000 和 020000000000，它们都表示十进制值-2147483648 ( $-2^{31}$ )。十六进制值 0xffffffff 和八进制值 037777777777 都表示十进制值-1。

如果 int 类型的十进制值大于 2147483648 ( $2^{31}$ )，或者如果值 2147483648 不是作为一元“-”运算符的操作数出现在任何地方，或者如果十六进制或八进制 int 值不适合 32 位，那么就会发生编译时错误。

int 值的示例：

```
0 2 0372 0xDadaCafe 1996 0x00FF00FF
```

long 类型最大的十进制值是 9223372036854775808L ( $2^{63}$ )。0L ~ 9223372036854775807L 之间的所有十进制值可以出现在 long 值可能出现的任何地方，但是值 9223372036854775808L 只能作为一元“非”运算符“-”的操作数出现。

long 类型最大的正十六进制值和八进制值分别是 0x7fffffffffffffffffL 和 07777777777777777777777777777777L，它们都等于 9223372036854775807L ( $2^{63}-1$ )。long 类型最小的负十六进制值和八进制值分别是 0x80000000000000000000L 和 01000000000000000000000000000000L，它们都等于十进制值 -9223372036854775808L ( $-2^{63}$ )。十六进制值 0xfffffffffffffffffL 和八进制值 01777777777777777777777777777777L 都表示十进制值-1L。

如果 long 类型的十进制值大于 9223372036854775808L ( $2^{63}$ )，或者如果值 9223372036854775808L 不是作为一元“-”运算符的操作数出现在任何地方，或者如果十六进制或八进制 long 值不适合 64 位，那么就会发生编译时错误。

long 值的示例：

```
01 0777L 0x1000000000L 2147483648L 0xC0B0L
```

### 3.10.2 浮点数

参见第4.2.3节，以了解有关浮点类型和值的一般性讨论。

浮点数具有以下部分：整数部分、十进制或十六进制小数点(用ASCII句点字符表示)、小数部分、指数和类型后缀。浮点数可以写作十进制值或十六进制值。对于十进制值，如果有指数，则用ASCII字母e或E来表示，后面可以选择接有带符号的整数。对于十六进制值，通常需要指数，并用ASCII字母p或P来表示，后面可以选择接有带符号的整数。

对于十进制浮点数，至少需要一位整数或小数部分，以及一个小数点、指数或浮点型后缀。所有其他部分都是可选的。对于十六进制浮点数，至少需要一位整数或小数部分，必须要有指数，而浮点型后缀是可选的。

如果一个浮点数带有ASCII字母F或f这样的后缀，那么它就是float类型；否则它就是double类型，并且可以选择带上ASCII字母D或d这样的后缀。

*FloatingPointLiteral:*

*DecimalFloatingPointLiteral*

*HexadecimalFloatingPointLiteral*

*DecimalFloatingPointLiteral:*

*Digits* . *Digits*<sub>opt</sub> *ExponentPart*<sub>opt</sub> *FloatTypeSuffix*<sub>opt</sub>  
*.* *Digits* *ExponentPart*<sub>opt</sub> *FloatTypeSuffix*<sub>opt</sub>  
*Digits* *ExponentPart* *FloatTypeSuffix*<sub>opt</sub>  
*Digits* *ExponentPart*<sub>opt</sub> *FloatTypeSuffix*

*ExponentPart:*

*ExponentIndicator* *SignedInteger*

*ExponentIndicator:* one of

*e*    *E*

*SignedInteger:*

*Sign*<sub>opt</sub> *Digits*

*Sign:* one of

*+*    *-*

*FloatTypeSuffix:* one of

*f*    *F*    *d*    *D*

*HexadecimalFloatingPointLiteral:*

*HexSignificand* *BinaryExponent* *FloatTypeSuffix*<sub>opt</sub>

*HexSignificand:*

*HexNumeral*

*HexNumeral* .

*0x* *HexDigits*<sub>opt</sub> . *HexDigits*

*0X* *HexDigits*<sub>opt</sub> . *HexDigits*

*BinaryExponent:*

*BinaryExponentIndicator* *SignedInteger*

*BinaryExponentIndicator:* one of

*p*    *P*

`float` 和 `double` 类型的元素是那些可以分别使用 IEEE 754 32 位单精度和 64 位双精度二进制浮点格式表示的值。

本规范为包 `java.lang` 的 `Float` 类和 `Double` 类的 `valueOf` 方法详细描述了从浮点数的 Unicode 字符串表示到内部 IEEE 754 二进制浮点表示的正确输入转换。

最大的正有穷 `float` 值是 `3.4028235e38f`。 `float` 类型最小的正有穷非 0 值是 `1.40e-45f`。最大的正有穷 `double` 值是 `1.7976931348623157e308`。 `double` 类型最小的正有穷非 0 值是 `4.9e-324`。

如果非 0 浮点值过大，使得在四舍五入转换成其内部表示时，它变成了一个 IEEE 754 无穷大的数，那么就会发生编译时错误。程序通过使用诸如 `1f/0f` 或 `-1d/0d` 之类的常量转换，或者通过使用 `Float` 类和 `Double` 类预定义的常量 `POSITIVE_INFINITY` 和 `NEGATIVE_INFINITY`，可以表示无穷大的数，而不会产生编译时错误。

如果非 0 浮点值过小，使得在四舍五入转换成其内部表示时，它变为 0，那么就会发生编译时错误。如果非 0 浮点值很小，但是在四舍五入转换成其内部表示时，它变成了一个非 0 的不正常的数字，那么将不会发生编译时错误。

用于表示非数字值的预定义常量在类 `Float` 和 `Double` 中被定义为 `Float.NaN` 和 `Double.NaN`。

`float` 值的示例：

```
1e1f2.f.3f0f3.14f6.022137e+23f
```

`double` 值的示例：

```
1e12..30.03.141e-9d1e137
```

除了用十进制和十六进制表示浮点值外，`Float` 类的 `intBitsToFloat` 方法和 `Double` 类的 `longBitsToDouble` 方法提供了一种依据十六进制或八进制整数值表示浮点值的方式。例如：

```
Double.longBitsToDouble(0x400921FB54442D18L)
```

的值等于 `Math.PI` 的值。

### 3.10.3 布尔值

`boolean` 类型具有两个值，分别用值 `true` 和 `false` 表示，它们都是用 ASCII 字母构成的。

布尔值总是 `boolean` 类型。

*BooleanLiteral: one of*  
`true false`

### 3.10.4 字符值

字符值被表示成字符或转义序列，它们被封闭在 ASCII 单引号中（单引号或省略号字符是 `\u0027`）。字符值只能表示 UTF-16 代码单元（3.1 节），即它们被限制为 `\u0000~\uffff` 之间的值。增补字符必须被表示成字符序列内的一个代理对（surrogate pair），或者一个整数，这取决于使用它们的 API。

字符值总是 `char` 类型。

*CharacterLiteral:*

*'SingleCharacter'*  
*'EscapeSequence'*

*SingleCharacter:*

*InputCharacter* but not `'` or `\`

第 3.10.6 节中描述了转义序列。

如第 3.4 节中所规定的那样，字符 `CR` 和 `LF` 永远不会是一个 *InputCharacter*；它们被

认可为构成一个 *LineTerminator*。

如果 *SingleCharacter* 或 *EscapeSequence* 后面接着一个不同于 “'” 的字符，则会出现编译时错误。

如果在开始 “'” 后面和结束 “'” 前面出现一个行终止符，则会出现编译时错误。

下面是几个 char 值的示例：

```
'a'
'8'
'\t'
'\'
'\'
'\u03a9'
'\uFFFF'
'\177'
'Ω'
'⊗'
```

由于Unicode转义处理的非常早，所以把其值为换行（LF）的字符值书写为 ‘\u000a’ 是不正确的；Unicode转义 \u000a 将会在转换步骤（1）（3.3节）中转换成实际的换行，并且该换行将在步骤（2）（3.4节）中变成一个 *LineTerminator*，因此字符值在步骤（3）中无效。相反，人们应该使用转义序列 ‘\n’（3.10.6节）。同样，把其值为回车（CR）的字符值书写为 ‘\u000d’ 也是不正确的。相反，应该使用 ‘\r’。

在C和C++中，字符值可以包含多个字符的表示，但是，这样一个字符的值是实现定义的。在Java编程语言中，字符值总是精确地表示一个字符。

### 3.10.5 字符串值

字符串值包含0个或多个封闭在双引号中的字符。字符可以通过转义序列表示——用于字符的一个转义序列的范围在U+0000~U+FFFF之间，用于字符的UTF-16代理代码单元的两个转义序列的范围在U+010000~U+10FFFF之间。

字符串值总是String类型（4.3.3节）。字符串值总是引用String类的相同实例（4.3.1节）。

*StringLiteral*:

" *StringCharacters<sub>opt</sub>* "

*StringCharacters*:

*StringCharacter*

*StringCharacters StringCharacter*

*StringCharacter*:

*InputCharacter* but not " or \

*EscapeSequence*

第3.10.6节中描述了转义序列。

如第3.4节中所规定的那样，字符CR和LF永远不会被看作是一个 *InputCharacter*；它们被认可为构成一个 *LineTerminator*。

如果在开始 “” 后面和结束 “” 前面出现一个行终止符，则会出现编译时错误。

总是可以把较长的字符串值分解为几段较短的部分，并使用字符串串联运算符 “+” 写成（可能要加括号）一个表达式（15.18.1节）。

下面是几个字符串值的示例：

```
"" //the empty string
"\ " //a string containing " alone
"This is a string" //a string containing 16 characters
"This is a " + //actually a string-valued constant expression,
    "two-line string" //formed from two string literals
```

由于Unicode转义处理的非常早，所以把包含单个换行（LF）的字符串值书写为 “\u000a” 是不正确的：Unicode转义\u000a将会在转换步骤（1）（3.3节）中转换成实际的换行，并且该换行将在步骤（2）（3.4节）中变成一个*LineTerminator*，因此字符串值在步骤（3）中无效。相反，人们应该书写 “\n”（3.10.6节）。同样，为包含单个回车（CR）的字符串值而书写 “\u000d” 也是不正确的。相反，应该使用 “\r”。

每个字符串值都是对String类（4.3.3节）的一个实例（4.3.1节、12.5节）的引用（4.3节）。String对象具有一个常量值。使用方法String.intern，使得字符串值——或者更一般地讲，作为常量表达式（15.28节）的值的字符串——都是被“限定的”，以共享独特的实例。

因此，包含编译单元的测试程序如下（7.3节）：

```
package testPackage;
class Test {
    public static void main(String[] args) {
        String hello = "Hello", lo = "lo";
        System.out.print((hello == "Hello") + " ");
        System.out.print((Other.hello == hello) + " ");
        System.out.print((other.Other.hello == hello) + " ");
        System.out.print((hello == ("Hel"+"lo")) + " ");
        System.out.print((hello == ("Hel"+lo)) + " ");
        System.out.println(hello == ("Hel"+lo).intern());
    }
}
class Other { static String hello = "Hello"; }
```

编译单元如下：

```
package other;
public class Other { static String hello = "Hello"; }
```

产生的输出如下：

```
true true true true false true
```

该示例阐释了6个要点：

- 相同包（第7章）中相同类（第8章）内的字符串值表示引用相同的String对象（4.3.1节）。
- 相同包中不同类内的字符串值表示引用相同的String对象。



- 不同包中不同类内的字符串值也表示引用相同的String对象。
- 通过常量表达式（15.28节）计算的字符串是在编译时计算的，然后将它们视为值。
- 通过串联在运行时计算的字符串是最新创建的，因此是截然不同的。
- 显式限定计算过的字符串所带来的结果是：与任何预先存在的字符串值相同的字符串具有相同的内容。

### 3.10.6 字符值和字符串值的转义序列

字符和字符串转义序列允许表示一些非图形字符以及字符值（3.10.4节）和字符串值（3.10.5节）中的单引号、双引号和反斜杠字符。

*EscapeSequence:*

```
\ b      /* \u0008: 退格BS */
\ t      /* \u0009: 水平制表符HT */
\ n      /* \u000a: 换行LF */
\ f      /* \u000c: 换页FF */
\ r      /* \u000d: 回车CR */
\ "      /* \u0022: 双引号" */
\ '      /* \u0027: 单引号' */
\ \      /* \u005c: 反斜杠\ */
```

*OctalEscape* /\* \u0000~\u00ff: 从八进制值 \*/

*OctalEscape:*

*\ OctalDigit*

*\ OctalDigit OctalDigit*

*\ ZeroToThree OctalDigit OctalDigit*

*OctalDigit: one of*

0 1 2 3 4 5 6 7

*ZeroToThree: one of*

0 1 2 3

如果在一个转义序列中，接在反斜杠后面的字符不同ASCII字符b、t、n、f、r、\*、'、\、0、1、2、3、4、5、6或7，则会出现编译时错误。Unicode转义\u会得到更早的处理（3.3节）（提供八进制转义是为了与C兼容，但是只能表示Unicode值\u0000~\u00FF，因此通常优先选择Unicode转义）。

### 3.10.7 空值

空类型具有一个值，即空引用，它是由ASCII字符构成的值null表示的。空值总是空类型。

*NullLiteral:*

null

### 3.11 分隔符

下面9个ASCII字符是分隔符（标点）：

*Separator: one of*

( ) { } [ ] ; , .

### 3.12 运算符

下面37个标记是运算符，它们是由ASCII字符构成的：

*Operator: one of*

= > < ! ~ ? :  
 == <= >= != && || ++ --  
 + - \* / & | ^ % << >> >>>  
 += -= \*= /= &= |= ^= %= <<= >>= >>>=

她的心肠就像铁石一样硬，您还是不用送她什么礼物，就送些像钻石似的硬货给她吧。

——威廉·莎士比亚，《维洛那二绅士》第一幕第一场

“拜访这些议员时，你是不自由的，因为这些议员的权力您我都能看到。”

——威廉·莎士比亚，《爱的徒劳》第五幕第二场

你，你，拉山德，你写诗句给我的孩子，和她交换着爱情的纪念物。

——威廉·莎士比亚，《仲夏夜之梦》第一幕第一场

这儿是一封从赫卡柏王后写来的信，还有她的女儿，我的爱人，给我的一件礼物……

——威廉·莎士比亚，《特洛伊罗斯和克瑞西达》第五幕第一场

有没有其他约定呢……？

——威廉·莎士比亚，《一报还一报》第四幕第一场

安静，亲爱的，不要害怕，亲爱的，狮子今晚睡着了。

——Luigi Creatore, George David Weiss, and Hugo E. Peretti

## 类型、值和变量

我不派出任何代理人或中介人，不提供价值代用品，而是提供价值本身。  
——沃尔特·惠特曼，《草叶集》

Java 编程语言是一种强类型化的语言，这意味着每个变量和每个表达式都有一个在编译时已知的类型。类型限制了变量（4.12 节）可以保存或者表达式可以产生的值，限制了那些值上支持的操作，并且确定了操作的含义。强类型有助于在编译时检测错误。

Java 编程语言的类型分为两类：基本类型和引用类型。基本类型（4.2 节）有布尔（boolean）型和数值型。数值型有整型 byte、short、int、long 和 char，以及浮点型 float 和 double。引用类型（4.3 节）有类类型、接口类型和数组类型。还有一个特殊的空类型。对象（4.3.1 节）是动态创建的类类型的实例或者动态创建的数组。引用类型的值是对对象的引用。所有对象（包括数组）都支持 Object 类（4.3.2 节）的方法。字符串值通过 String 对象（4.3.3 节）表示。

类型存在于编译时。有些类型对应于在运行时存在的类和接口。类型与类或接口之间的对应关系是不完全的，这有两个原因：

（1）在运行时，类和接口是通过 Java 虚拟机使用类加载器加载的。每个类加载器定义了它自己的类和接口的集合。因此，两个加载器有可能加载同一个类或接口定义，但却在运行时产生了不同的类或接口。

（2）类型参数和类型变量（4.4 节）不是在运行时具体化的。因此，不同的参数化类型（4.5 节）是由相同的类或接口在运行时实现的。的确，给定泛型（generic）类型声明（8.1.2 节、9.1.2 节）的所有调用都共享单独一个运行时实现。

原因（1）的后果是：如果加载代码的类加载器不一致，那么正确编译的代码可能在连接时出错。参见 Sheng Liang 和 Gilad Bracha 在《Proceedings of OOPSLA '98》中撰写的论文《Dynamic Class Loading in the Java™ Virtual Machine》（该论文发表为《ACM SIGPLAN Notices》第 33 卷，第 10 册，1998 年 10 月，第 36~44 页）和《The Java™ Virtual Machine Specification, Second Edition》，以获取详细信息。

原因（2）的后果是：堆污染（4.12.2.1 节）的可能性。在某些条件下，有可能出现一个参数化类型的变量引用一个并非那种参数化类型的对象。该变量将总是引用一个对象，

这个对象是实现那种参数化类型的类的一个实例。有关进一步的讨论，参见第 4.12.2 节。

## 4.1 各种类型和值

Java 编程语言中有两种类型：基本类型（4.2 节）和引用类型（4.3 节）。相应地有两种数据值，即基本值（4.2 节）和引用值（4.3 节），它们可以存储在变量中，作为参数传递，被方法返回，并在其上执行各种操作。

*Type:*

*PrimitiveType*

*ReferenceType*

还有一种特殊的空类型，即表达式 `null` 的类型，它没有名称。由于空类型没有名称，所以不可能声明一个空类型的变量，或者强制转换为空类型。空引用是空类型的表达式惟一可能的值。空引用总是可以强制转换为任何引用类型。实际上，程序员可以忽略空类型，并假定 `null` 只是一个可以为任何引用类型的特殊值。

## 4.2 基本类型和值

基本类型是由 Java 编程语言预定义的，并通过其保留的关键字（3.9 节）命名：

*PrimitiveType:*

*NumericType*

`boolean`

*NumericType:*

*IntegralType*

*FloatingPointType*

*IntegralType:one of*

`byte short int long char`

*FloatingPointType:one of*

`float double`

基本值不与其他基本值共享状态。其类型为基本类型的变量总是保存具有相同类型的基本值。基本类型的变量的值只能通过该变量上的赋值运算进行更改[包括递增（15.14.2 节、15.15.1 节）和递减（15.14.3 节、15.15.2 节）运算符]。

数值类型有整型和浮点型。

整型有 `byte`、`short`、`int` 和 `long`，它们的值分别是 8 位、16 位、32 位和 64 位带符号 2 的补码的整数；整型还包括 `char`，它的值是 16 位无符号整数，表示 UTF-16 代码单元（3.1 节）。

浮点型有 `float`，它的值包括 32 位 IEEE 754 浮点数；还有 `double`，它的值包括 64 位 IEEE 754 浮点数。

boolean 类型正好有两个值: true 和 false。

### 4.2.1 整型和值

整型的值都是以下范围内的整数:

- 对于 byte, 为-128~127 (含-128 和 127)。
- 对于 short, 为-32768~32767 (含-32768 和 32767)。
- 对于 int, 为-2147483648~2147483647 (含-2147483648 和 2147483647)。
- 对于 long, 为-9223372036854775808~9223372036854775807 (含-9223372036854775808 和 9223372036854775807)。
- 对于 char, 为'\u0000'~'\uffff' (含'\u0000' 和 '\uffff'), 即从 0~65535。

### 4.2.2 整数运算

Java 编程语言提供了许多作用于整数值的运算符:

- 比较运算符, 产生 boolean 类型的值:
  - ◆ 数值比较运算符: <、<=、> 和 >= (15.20.1 节)。
  - ◆ 数值相等性运算符: == 和 != (15.21.1 节)。
- 数值运算符, 产生 int 或 long 类型的值:
  - ◆ 一元加、减运算符+和- (15.15.3 节、15.15.4 节)。
  - ◆ 乘法运算符\*、/和% (15.17 节)。
  - ◆ 加法运算符+和- (15.18 节)。
  - ◆ 递增运算符++, 同为前缀 (15.15.1 节) 和后缀 (15.14.2 节)。
  - ◆ 递减运算符--, 同为前缀 (15.15.2 节) 和后缀 (15.14.3 节)。
  - ◆ 带符号和无符号移位运算符<<、>>和>>> (15.19 节)。
  - ◆ 逐位求补运算符~ (15.15.5 节)。
  - ◆ 整数逐位运算符&、|和^ (15.22.1 节)。
- 条件运算符?: (15.25 节)
- 强制转换运算符, 可以把一个整型值转换成任何指定数值类型的值 (5.5 节、15.16 节)。
- 字符串串接运算符+(15.18.1 节), 当给定一个 String 操作数和一个整型操作数时, 该运算符将把整型操作数转换成以十进制形式表示其值的 String, 然后产生一个最新创建的 String, 它是两个字符串的串接。

类 Byte、Short、Integer、Long 和 Character 中预定义了其他有用的构造函数、方法和常量。

如果一个整型运算符 (而不是一个移位运算符) 具有至少一个 long 类型的操作数, 那么, 将使用 64 位的精度执行运算, 并且数值运算符的结果是 long 类型。如果另一个操作数不是 long 类型, 则首先会通过数值提升 (5.6 节) 将其加宽到类型 long。否则, 将使用 32 位的精度执行运算, 并且数值运算符的结果是 int 类型。如果任一个操作数不是 int 类型, 则首先会通过数值提升将其加宽到 int 类型。

内置的整型运算符不会以任何方式指示上溢或下溢。如果需要空引用的拆箱转换（5.1.8节），则整型运算符可以抛出一个 `NullPointerException`。此外，可以抛出一个异常（第11章）的惟一整型运算符是：整型除法运算符 `/`（15.17.2节）和整型求余运算符 `%`（15.17.3节），如果右边的操作数为0，那么它们会抛出一个 `ArithmeticException`；以及递增和递减运算符 `++`（15.15.1节、15.15.2节）和 `--`（15.14.3节、15.14.2节），如果需要装箱转换（5.1.7节），并且没有足够的内存可用于执行该转换，那么它们会抛出一个 `OutOfMemoryError`。

示例：

```
class Test {
    public static void main(String[] args) {
        int i = 10000000;
        System.out.println(i * i);
        long l = i;
        System.out.println(l * l);
        System.out.println(20296 / (l - i));
    }
}
```

产生如下输入：

```
-727379968
10000000000000
```

并且随后会由于 `l-i` 而在除法中遇到一个 `ArithmeticException`，因为 `l-i` 等于0。第一个乘法是以32位的精度执行的，而第二个乘法是一个 `long` 乘法。值 `-727379968` 是数学结果 `10000000000000` 的低32位的十进制值，`10000000000000` 对于类型 `int` 是一个过大的值。

任何整型的任何值都可以被强制转换为任何数值类型，反之亦然。在整型和 `boolean` 类型之间没有强制转换。

### 4.2.3 浮点型、格式和值

浮点型有 `float` 和 `double`，从概念上讲，它们与单精度32位和双精度64位格式 IEEE 754 值和运算相关联，如 *IEEE Standard for Binary Floating-Point Arithmetic*，ANSI/IEEE Standard 754-1985（IEEE，New York）中所规定的那样。

IEEE 754 标准不仅包括由符号和数量组成的正数和负数，还包括正0、负0、正无穷大、负无穷大和特殊的非数值（Not-a-Number，NaN）。NaN 值用于表示某些无效操作（如0除以0）的结果。`float` 和 `double` 类型的 NaN 常量被预定义为 `Float.NaN` 和 `Double.NaN`。

Java 编程语言的每种实现都需要支持浮点值的两个标准集，称为浮点值集合和双精度值集合。此外，Java 编程语言的实现可以支持两种扩展指数的浮点值集中的任何一个，或者两个都支持，它们是浮点扩展指数的值集合和双精度扩展指数的值集。在某些情况下，可以使用这些扩展指数的值集代替标准值集来表示 `float` 或 `double` 类型的表达式的值（5.1.13节、15.4节）。

任何浮点值集的有穷非0值都可以表示成  $s \cdot m \cdot 2^{(e-N+1)}$  的形式，其中  $s$  为+1或-1， $m$  是一个小于  $2^N$  的正整数， $e$  是  $E_{min} = -(2^{K-1}-2)$  和  $E_{max} = 2^{K-1}-1$  之间（含  $E_{min}$  和  $E_{max}$ ）的一个整数，其中  $N$



和 $K$ 是依赖于值集的参数。有些值可以多种方式表示成这种形式：例如，假设可以使用 $s$ 、 $m$ 和 $e$ 的某些值将值集中的一个值 $v$ 表示成这种形式，这样如果碰巧 $m$ 是偶数并且 $e$ 小于 $2^{K-1}$ ，则可以通过把 $m$ 减半并把 $e$ 增1来产生同一个值 $v$ 的第二种表示。如果 $m \geq 2^{(N-1)}$ ，则把这种形式的表示称为规范化；否则，将表示称为非规范化。如果值集中的某个值不能用 $m \geq 2^{(N-1)}$ 的方式表示，那么就称该值为非规范化的值，因为它不具有规范化的表示。

表4.1总结了对于两个需要的和两个可选的浮点值集，参数 $N$ 和 $K$ （以及派生参数 $E_{min}$ 和 $E_{max}$ ）上的约束条件。

表 4.1 浮点值集参数

参数	浮点数	浮点扩展的指数	双精度数	双精度扩展的指数
$N$	24	24	53	53
$K$	8	$\geq 11$	11	$\geq 15$
$E_{max}$	+127	$\geq +1023$	+1023	$\geq +16383$
$E_{min}$	-126	$\leq -1022$	-1022	$\leq -16382$

其中一个或两个可扩展的指数值集受到一种实现的支持，则对于每个受支持的可扩展的指数值集，有一个依赖于特定实现的常量 $K$ ，它的值受到表4.1的约束；这个 $K$ 值反过来又规定了 $E_{min}$ 和 $E_{max}$ 的值。

4个值集中的每一个不仅包括上述归属于它的有穷非0值，还包括NaN值和以下4个值：正0、负0、正无穷大和负无穷大。

注意，表4.1中的约束条件被设计成使得浮点值集中的每个元素同时还必须是浮点扩展的指数值集、双精度值集和双精度扩展的指数值集的元素。同样，双精度值集中的每个元素也必须是双精度扩展的指数值集的元素。与对应的标准值集相比，每个扩展的指数值集具有更大的指数值范围，但不具有更高的精度。

浮点值集的元素严格地是可以使用IEEE 754标准中定义的单精度浮点格式表示的值。双精度值集的元素严格地是可以使用IEEE 754标准中定义的双精度浮点格式表示的值。但是要注意，这里定义的浮点扩展的指数值集和双精度扩展的指数值集的元素并不分别对应于可以使用IEEE 754单精度扩展的格式和双精度扩展的格式表示的值。

浮点值集、浮点扩展的指数值集、双精度值集和双精度扩展的指数值集都不是一种类型。Java编程语言的实现使用浮点值集的元素表示float类型的值总是正确的；但是，在某些代码区域，可能允许一个实现使用浮点扩展的指数值集的元素作为替代。同样，一个实现使用双精度值集的元素表示double类型的值总是正确的；但是在某些区域，可能允许一个实现使用双精度扩展的指数值集的元素作为替代。

除了NaN之外，浮点值都是有序的；按照从最小到最大的顺序排列，它们是负无穷大、负有穷非0值、正0和负0、正有穷非0值和正无穷大。

IEEE 754允许多个不同NaN值中的每一个都具有其单精度和双精度浮点格式。虽然在生成一个新的NaN时，每种硬件体系结构会返回针对该NaN的一个特殊的位模式，但是程序员也可以用不同的位模式创建NaN，例如，编码回溯性的诊断信息。

在极大程度上, Java 平台把给定类型的 NaN 值看作好像折叠进单精度规范值中(因此, 本规范通常引用任意的 NaN, 就像是引用规范值一样)。但是 Java 平台 1.3 版引入了使程序员能够区分 NaN 值的方法: `Float.floatToRawIntBits` 和 `Double.doubleToRawLongBits` 方法。有兴趣的读者可以参阅 `Float` 和 `Double` 类的规范, 以获取更多信息。

对正 0 和负 0 执行比较运算的结果相等; 因此, 表达式 `0.0==-0.0` 的结果为 `true`, 而 `0.0>-0.0` 的结果为 `false`。但是, 其他运算可以区分正 0 和负 0; 例如, `1.0/0.0` 的值是正无穷大, 而 `1.0/-0.0` 的值是负无穷大。

NaN 是无序的, 因此如果任何一个或两个操作数为 NaN, 那么数值比较运算符 `<`、`<=`、`>` 和 `>=` 会返回 `false` (15.20.1 节)。如果任何一个操作数是 NaN, 那么相等性运算符 `==` 会返回 `false`, 不等性运算符 `!=` 会返回 `true` (15.21.1 节)。特别地, 当且仅当 `x` 是 NaN 时, `x!=x` 为 `true`; 如果 `x` 或 `y` 是 NaN, 那么 `(x<y)==!(x>y)` 将为 `false`。

任何浮点型的值都可以强制转换为任何数值类型, 反之亦然。在浮点型和 `boolean` 类型之间不能进行强制转换。

#### 4.2.4 浮点运算

Java 编程语言提供了许多可以作用于浮点值的运算符:

- 比较运算符, 产生 `boolean` 类型的值:
  - ◆ 数值比较运算符 `<`、`<=`、`>` 和 `>=` (15.20.1 节)
  - ◆ 数值相等性运算符 `==` 和 `!=` (15.21.1 节)
- 数值运算符, 产生 `float` 或 `double` 类型的值:
  - ◆ 一元加、减运算符 `+` 和 `-` (15.15.3 节、15.15.4 节)
  - ◆ 乘法运算符 `*`、`/` 和 `%` (15.17 节)
  - ◆ 加法运算符 `+` 和 `-` (15.18.2 节)
  - ◆ 递增运算符 `++`, 同为前缀 (15.15.1 节) 和后缀 (15.14.2 节)
  - ◆ 递减运算符 `--`, 同为前缀 (15.15.2 节) 和后缀 (15.14.3 节)
- 条件运算符 `?:` (15.25 节)
- 强制转换类型转换运算符, 可以把一个浮点值转换成任何指定数值类型的值 (5.5 节、15.16 节)。
- 字符串串接运算符 `+` (15.18.1 节), 当给定一个 `String` 操作数和一个浮点操作数时, 该运算符将把浮点操作数转换成以十进制形式表示其值的 `String` (而不会丢失信息), 然后通过串接两个字符串产生一个最新创建的 `String`。

类 `Float`、`Double` 和 `Math` 中预定义了其他有用的构造函数、方法和常量。

如果二元运算符至少有一个操作数是浮点型, 那么该运算就是一个浮点运算, 即使另一个操作数是整数也是如此。

如果数值运算符至少有一个操作数是 `double` 类型, 那么将使用 64 位浮点运算执行该运算, 并且数值运算符的结果是 `double` 类型的值[如果另一个操作数不是 `double` 类型, 则首先会通过数值提升 (5.6 节) 将其加宽到 `double` 类型]。否则, 将使用 32 位浮

点运算执行该运算，并且数值运算符的结果是 `float` 类型。如果另一个操作数不是 `float` 类型，则首先会通过数值提升将其加宽到 `float` 类型。

IEEE 754 规定了浮点数上的运算符的行为方式[除求余运算符 (15.17.3 节) 之外]。特别地，Java 编程语言需要支持 IEEE 754 非规范化的浮点数和逐渐下溢 (gradual underflow)，这使得证明人们期待的特殊数值算法的性质更容易。如果计算结果是一个非规范数，则浮点运算不会刷新为 0 (flush to zero)。

Java 编程语言要求浮点运算按如下方式工作：就像每个浮点运算符将其浮点运算结果四舍五入为结果精度一样。不精确的结果必须被四舍五入为最接近无限精确结果的有代表性的值；如果两个最接近的有代表性的值近似相等，那么将会选择其最低有效位为 0 的那个值。这是 IEEE 754 标准默认的四舍五入模式，称为四舍五入至最接近的值。

当把浮点值转换为整数时，Java 语言使用四舍五入到 0 (5.1.3 节)，在这种情况下，就好像把数截断一样，丢弃尾数位。选择四舍五入到 0，可以使得这种格式的值在数量上最接近于并且不大于无限精确的结果。

如果需要空引用的拆箱转换 (5.1.8 节)，浮点运算符可能抛出一个 `NullPointerException`。此外，可以抛出一个异常 (第 11 章) 的惟一的浮点运算符是递增和递减运算符 `++` (15.15.1 节、15.15.2 节) 和 `--` (15.14.3 节、15.14.2 节)，如果需要装箱转换 (5.1.7 节)，并且没有足够的内存可用于执行该转换，那么它们会抛出一个 `OutOfMemoryError`。

上溢运算会产生一个带符号的无穷大，下溢运算会产生一个非规范的值或者一个带符号的 0，不具有数学限定结果的运算会产生一个 NaN。把 NaN 作为一个操作数的所有数值运算都会产生一个 NaN 作为结果。正如已经描述过的，NaN 是无序的，因此，涉及一个或两个 NaN 的数值比较运算会返回 `false`，而涉及 NaN 的任何 `!=` 比较运算都会返回 `true`，当 `x` 是 NaN 时，包括 `x!=x`。

示例程序：

```
class Test {
    public static void main(String[] args) {
        // An example of overflow:
        double d = 1e308;
        System.out.print("overflow produces infinity: ");
        System.out.println(d + "**10==" + d*10);
        // An example of gradual underflow:
        d = 1e-305 * Math.PI;
        System.out.print("gradual underflow: " + d + "\n");
        for (int i = 0; i < 4; i++)
            System.out.print(" " + (d /= 100000));
        System.out.println();
        // An example of NaN:
        System.out.print("0.0/0.0 is Not-a-Number: ");
        d = 0.0/0.0;
        System.out.println(d);
        // An example of inexact results and rounding:
        System.out.print("inexact results with float:");
        for (int i = 0; i < 100; i++) {
```

```

        float z = 1.0f / i;
        if (z * i != 1.0f)
            System.out.print(" * + i);
    }
    System.out.println();
    // Another example of inexact results and rounding:
    System.out.print("inexact results with double:");
    for (int i = 0; i < 100; i++) {
        double z = 1.0 / i;
        if (z * i != 1.0)
            System.out.print(" * + i);
    }
    System.out.println();
    // An example of cast to integer rounding:
    System.out.print("cast to int rounds toward 0: ");
    d = 12345.6;
    System.out.println((int)d + " * + (int)(-d));
}
}

```

产生如下输出:

```

overflow produces infinity: 1.0e+308*10==Infinity
gradual underflow: 3.141592653589793E-305
3.1415926535898E-310 3.141592653E-315 3.142E-320 0.0
0.0/0.0 is Not-a-Number: NaN
inexact results with float: 0 41 47 55 61 82 83 94 97
inexact results with double: 0 49 98
cast to int rounds toward 0: 12345 -12345

```

值得一提的是, 这个示例证实逐渐下溢可能导致精度逐渐损失。

当  $i$  为 0 时, 结果涉及到除以 0, 因此  $z$  变为正无穷大, 并且  $z*0$  是 NaN, 它不等于 1.0。

## 4.2.5 boolean 类型和 boolean 值

boolean 类型表示一个逻辑量, 它具有两个可能的值, 通过值 true 和 false (3.10.3 节) 来指示。布尔运算符有:

- 关系运算符 == 和 != (15.21.2 节)
  - 逻辑求补运算符 ! (15.15.6 节)
  - 逻辑运算符 &、^ 和 | (15.22.2 节)
  - “条件与”运算符 && (15.23 节) 和 “条件或”运算符 || (15.24 节)
  - 条件运算符 ?: (15.25 节)
  - 字符串串接运算符 + (15.18.1 节), 当给定一个 String 操作数和一个布尔操作数时, 该运算符将把布尔操作数转换成 String (“true” 或 “false”), 然后产生一个最新创建的 String, 它是两个字符串的串接
- 布尔表达式确定了多种语句中的控制流程:

- if 语句 (14.9 节)
- while 语句 (14.12 节)
- do 语句 (14.13 节)
- for 语句 (14.14 节)

boolean 表达式还确定了哪个子表达式是通过条件?:运算符 (15.25 节) 求值的。

只有 boolean 或 Boolean 表达式可以用于控制流程语句中, 并且作为条件运算符?: 的第一个操作数。依据 C 语言的约定, 任何非 0 值都是 true, 通过表达式  $x \neq 0$ , 可以把整数  $x$  转换成一个 boolean。依据 C 语言的约定, 不同于 null 的任何引用都是 true, 通过表达式  $obj \neq \text{null}$ , 可以把对象引用  $obj$  转换成一个 boolean。

允许把 boolean 值强制转换成类型 boolean 或 Boolean (5.1.1 节); 不允许在类型 boolean 上执行其他强制转换。可以通过字符串转换把 boolean 转换成一个字符串 (5.4 节)。

## 4.3 引用类型和值

有三种引用类型: 类类型 (第 8 章)、接口类型 (第 9 章) 和数组类型 (第 10 章)。可以用类型参数 (4.4 节) 对引用类型进行参数化 (4.5 节)。

*ReferenceType:*

*ClassOrInterfaceType*

*TypeVariable*

*ArrayType*

*ClassOrInterfaceType:*

*ClassType*

*InterfaceType*

*ClassType:*

*TypeDeclSpecifier* *TypeArguments*<sub>opt</sub>

*InterfaceType:*

*TypeDeclSpecifier* *TypeArguments*<sub>opt</sub>

*TypeDeclSpecifier:*

*TypeName*

*ClassOrInterfaceType* . *Identifier*

*TypeName:*

*Identifier*

*TypeName* . *Identifier*

*TypeVariable:*

*Identifier*



*ArrayType:*

*Type []*

类或接口类型包含类型声明指定符；可选择后接类型参数（在这种情况下，它是参数化类型）。第 4.5.1 节中描述了类型参数。

类型声明指定符可以是类型名称（6.5.5 节），或者是类或接口类型，后接“.”和一个标识符。在后一种情况下，指定符具有 *T.id* 形式，其中 *id* 必须是一个可访问（6.6 节）成员类型（8.5 节、9.5 节）*T* 的简单名称，否则将会发生编译时错误。指定符指明了成员类型。

示例代码：

```
class Point { int[] metrics; }  
interface Move { void move(int deltax, int deltay); }
```

声明一个类类型 *Point*，一个接口类型 *Move*，并使用一个数组类型 *int[]*（*int* 数组）来声明类 *Point* 的字段 *metrics*。

### 4.3.1 对象

对象是一个类的实例或一个数组。

引用值（通常只称为引用）是指向这些对象的指针，以及不指向任何对象的特殊的空引用。

类实例是通过类实例创建表达式（15.9 节）显式创建的。数组是通过数组创建表达式（15.10 节）显式创建的。

当在非常量（15.28 节）表达式中使用字符串串接运算符+（15.18.1 节）时，会隐式创建一个新的类实例，从而产生 *String* 类型（4.3.3 节）的一个新对象。当对一个数组初始化表达式（10.6 节）求值时，就会隐式创建一个新的数组对象；当初始化类或接口时（12.4 节），当创建一个新的类实例时（15.9 节），或者当执行一条局部变量声明语句时（14.4 节），就可能发生这种情况。可以通过装箱转换（5.1.7 节）隐式创建 *Boolean*、*Byte*、*Short*、*Character*、*Integer*、*Long*、*Float* 和 *Double* 类型的新对象。

下面的示例中阐释了许多这类情况：

```
class Point {  
    int x, y;  
    Point() { System.out.println("default"); }  
    Point(int x, int y) { this.x = x; this.y = y; }  
    // A Point instance is explicitly created at class initialization time:  
    static Point origin = new Point(0,0);  
    // A String can be implicitly created by a + operator:  
    public String toString() {  
        return "(" + x + "," + y + ")";  
    }  
}  
class Test {  
    public static void main(String[] args) {  
        // A Point is explicitly created using newInstance:  
        Point p = null;  
    }  
}
```



```
try {
    p = (Point)Class.forName("Point").newInstance();
} catch (Exception e) {
    System.out.println(e);
}
// An array is implicitly created by an array constructor:
Point a[] = { new Point(0,0), new Point(1,1) };
// Strings are implicitly created by + operators:
System.out.println("p: " + p);
System.out.println("a: { " + a[0] + ", " + a[1] + " }");

// An array is explicitly created by an array creation expression:
String sa[] = new String[2];
sa[0] = "he"; sa[1] = "llo";
System.out.println(sa[0] + sa[1]);
}
```

产生如下输出:

```
default
p: (0,0)
a: { (0,0), (1,1) }
hello
```

引用对象的运算符有:

- 字段访问, 使用限定名称 (6.6 节) 或字段访问表达式 (15.11 节)
- 方法调用 (15.12 节)
- 强制类型转换运算符 (5.5 节、15.16 节)
- 字符串串接运算符+ (15.18.1 节), 当给定一个 String 操作数和一个引用时, 该运算符将通过调用被引用对象的 toString 方法(如果引用或 toString 的结果是一个空引用, 则使用"null"), 把引用转换成 String, 然后产生一个最新创建的 String, 它是两个字符串的串接
- instanceof 运算符 (15.20.2 节)
- 引用相等性运算符==和!= (15.21.3 节)
- 条件运算符?: (15.25 节)

可能有许多引用指向同一个对象。大多数对象都有状态, 它们存储在对象的字段中, 这些对象是类的实例; 或者存储在变量中, 这些变量是数组对象的元素。如果两个变量包含指向同一个对象的引用, 就可以使用指向对象的一个变量的引用来修改对象的状态, 然后可以通过另一个变量中的引用来观察更改过的状态。

示例程序:

```
class Value { int val; }
class Test {
    public static void main(String[] args) {
        int i1 = 3;
        int i2 = i1;
        i2 = 4;
        System.out.print("i1==" + i1);
    }
}
```

```

        System.out.println(" but i2==" + i2);
        Value v1 = new Value();
        v1.val = 5;
        Value v2 = v1;
        v2.val = 6;
        System.out.print("v1.val==" + v1.val);
        System.out.println(" and v2.val==" + v2.val);
    }
}

```

产生如下输出：

```

i1==3 but i2==4
v1.val==6 and v2.val==6

```

这是由于 `v1.val` 和 `v2.val` 引用由惟一的 `new` 表达式创建的一个 `Value` 对象中相同的实例变量（4.12.3 节），虽然 `i1` 和 `i2` 是不同的变量。

参见第 10 章和第 15.10 节，以查看创建和使用数组的示例。

每个对象都有一个关联的锁（17.1 节），同步化方法（8.4.3 节）和同步化语句（14.19 节）使用这个锁来控制多个线程对状态的并发访问（第 17 章）。

### 4.3.2 Object 类

`Object` 类是所有其他类的超类（8.1 节）。`Object` 类型的变量可以存储一个指向空引用或任何对象的引用，无论该对象是类的实例还是一个数组（第 10 章）。所有的类和数组类型都继承 `Object` 类的方法，总结如下：

```

package java.lang;

public class Object {
    public final Class<?> getClass() { . . . }
    public String toString() { . . . }
    public boolean equals(Object obj) { . . . }
    public int hashCode() { . . . }
    protected Object clone()
        throws CloneNotSupportedException { . . . }
    public final void wait()
        throws IllegalMonitorStateException,
        InterruptedException { . . . }
    public final void wait(long millis)
        throws IllegalMonitorStateException,
        InterruptedException { . . . }
    public final void wait(long millis, int nanos) { . . . }
        throws IllegalMonitorStateException,
        InterruptedException { . . . }
    public final void notify() { . . . }
        throws IllegalMonitorStateException
    public final void notifyAll() { . . . }
        throws IllegalMonitorStateException
    protected void finalize()
        throws Throwable { . . . }
}

```

Object 的成员如下：

- 方法 `getClass` 返回 `Class` 对象，它表示对象的类。`Class` 对象为每个引用类型而存在。例如，可以使用它来发现类、它的成员、它的直接超类以及它实现的任何接口的完全限定名称。声明为 `synchronized` (8.4.3.6 节) 的类方法可以同步与类的 `Class` 对象关联的锁。方法 `Object.getClass()` 必须被 Java 编译器特殊对待。方法调用 `e.getClass()` 的类型（其中表达式 `e` 具有静态类型 `T`）是 `Class<? extends T>`。
- 方法 `toString` 返回对象的 `String` 表示。
- 方法 `equals` 和 `hashCode` 在诸如 `java.util.Hashtable` 之类的散列表中非常有用。方法 `equals` 定义了对象相等性的概念，它基于值（而不是引用）比较。
- 方法 `clone` 用于制作对象的副本。
- 方法 `wait`、`notify` 和 `notifyAll` 用于使用线程的并发编程中，如第 17 章所述。
- 在某个对象刚好要被销毁之前，会运行方法 `finalize`，如第 12.6 节所述。

### 4.3.3 String 类

`String` 类的实例表示 Unicode 字符序列。`String` 对象具有一个常量（不变的）值。字符串值（3.10.5 节）是对 `String` 类的实例的引用。

当结果不是一个编译时常量时（15.28 节），字符串串接运算符 `+`（15.18.1 节）会隐式创建一个新的 `String` 对象。

### 4.3.4 当引用类型相同时

如果两个引用类型具有相同的二进制名称（13.1 节），并且它们的类型参数（如果有的话）也相同，递归地应用这个定义，那么这两个引用类型就是相同的编译时类型。当两个引用类型相同时，有时把它们称为相同的类或相同的接口。

在运行时，可以通过不同的类加载器同时加载多个具有相同二进制名称的引用类型。这些类型可能或不可能表示相同的类型声明。即使两个这样的类型确实表示相同的类型声明，它们也被认为是有区别的。

如果两个引用类型满足以下条件，则它们是相同的运行时类型：

- 它们都是类或接口类型，都通过相同的类加载器定义，并且具有相同的二进制名称（13.1 节），在这种情况下，有时把它们称为相同的运行时类或相同的运行时接口。
- 它们都是数组类型，并且它们的元素类型是相同的运行时类型（第 10 章）。

## 4.4 类型变量

类型变量（4.4 节）是一种非限定性标识符。类型变量是由泛型类声明（8.1.2 节）、泛型接口声明（9.1.2 节）、泛型方法声明（8.4.4 节）以及泛型构造函数声明（8.8.4 节）引入

的。

*TypeParameter:*

*TypeVariable TypeBound<sub>opt</sub>*

*TypeBound:*

*extends ClassOrInterfaceType AdditionalBoundList<sub>opt</sub>*

*AdditionalBoundList:*

*AdditionalBound AdditionalBoundList*

*AdditionalBound*

*AdditionalBound:*

*& InterfaceType*

类型变量具有一个可选界限  $T \& I_1 \dots I_n$ 。这个界限包含一个类型变量，或者一个类或接口类型  $T$ ，其后可能接有更多的接口类型  $I_1, \dots, I_n$ 。如果没有指定类型变量的界限，就假定其为 `Object`。如果类型  $I_1 \dots I_n$  中的任何一个为类类型或类型变量，则会发生编译时错误。界限的所有成员类型的擦除（erasure）（4.6 节）必须两两不同，否则就会发生编译时错误。界限中的类型的顺序是惟一重要的因素，这是由于类型变量的擦除是由其界限中的第一个类型确定的，并且类类型或类型变量只可能出现在第一个位置。

对于由相同泛型接口的不同参数化形成的两个接口类型，类型变量不可能同时是这两个接口类型的子类型。

参见第 6.3 节，以了解定义类型变量作用域的规则。

具有界限  $T \& I_1 \dots I_n$  的类型变量  $X$  的成员是出现在声明类型变量那一点的交集类型（4.9 节） $T \& I_1 \dots I_n$  的成员。

#### 示例

下面的示例阐释了类型变量具有什么成员。

```
package TypeVarMembers;
```

```
class C {
    void mCDefault() {}
    public void mCPublic() {}
    private void mCPrivate() {}
    protected void mCProtected() {}
}
class CT extends C implements I {}
interface I {
    void mI();
    <T extends C & I> void test(T t) {
        t.mI(); // OK
        t.mCDefault(); // OK
        t.mCPublic(); // OK
        t.mCPrivate(); // compile-time error
    }
}
```

```
t.mCProtected(); // OK
```

```
}
```

```
}
```

类型变量  $T$  具有与交集类型  $C \& I$  相同的成员，交集类型  $C \& I$  反过来又具有与空类  $CT$  相同的成员，空类  $CT$  是在相同的作用域中用等价的超类型定义的。接口的成员总是公共类型，因此总是会被继承（除非被重写）。因此， $mI$  是  $CT$  和  $T$  的成员。在  $C$  的成员当中，除  $mCPrivate$  之外的所有成员都会被  $CT$  继承，因而它们是  $CT$  和  $T$  的成员。

如果在一个不同于  $T$  的包中声明  $C$ ，那么对  $mCDefault$  的调用将引发一个编译时错误，因为在声明  $T$  的那一点上不能访问该成员。

## 4.5 参数化类型

参数化类型包含一个类或接口名称  $C$ ，以及实际的类型参数列表  $\langle T_1, \dots, T_n \rangle$ 。如果  $C$  不是泛型类或接口的名称，或者如果实际类型参数列表中类型参数的数量不同于声明的  $C$  的类型参数的数量，那么就会发生编译时错误。在后面，无论何时我们提到一个类或接口类型，都包括了泛型版本，除非显式地排除了该版本。在本节中，设  $A_1, \dots, A_n$  是  $C$  的形式类型参数，设  $B_i$  是声明的  $A_i$  的界限。符号  $[A_i := T_i]$  表示用类型  $T_i$  代替类型变量  $A_i$ ，其中  $1 \leq i \leq n$ ，在本规范的所有部分都使用了这种约定。

设  $P = G \langle T_1, \dots, T_n \rangle$  是一种参数化类型。如果  $P$  服从导致类型  $G \langle X_1, \dots, X_n \rangle$  的捕获转换（5.1.10 节）的支配，那么对于每个实际的类型参数  $X_i$  ( $1 \leq i \leq n$ )，必定有  $X_i <: B_i[A_j := X_j, \dots, A_n := X_n]$ （4.10 节），否则就会发生一个编译时错误。

### 讨论

示例：参数化类型。

```
Vector<String>
```

```
Seq<Seq<A>>
```

```
Seq<String>.Zipper<Integer>
```

```
Collection<Integer>
```

```
Pair<String,String>
```

```
// Vector<int> -- illegal, primitive types cannot be arguments
```

```
// Pair<String> -- illegal, not enough arguments
```

```
// Pair<String,String,String> -- illegal, too many arguments
```

如果下面两个条件中的任何一个成立，则可证实两个参数化类型是截然不同的：

- 它们是截然不同的泛型类型声明的调用。
- 可证实任何一个类型参数是截然不同的。

### 4.5.1 类型参数和通配符

类型参数可以是引用类型或通配符。

*TypeArguments:*

*< ActualTypeArgumentList >*

*ActualTypeArgumentList:*

*ActualTypeArgument*

*ActualTypeArgumentList , ActualTypeArgument*

*ActualTypeArgument:*

*ReferenceType*

*Wildcard*

*Wildcard:*

*? WildcardBounds<sub>Opt</sub>*

*WildcardBounds:*

*extends ReferenceType*

*super ReferenceType*

#### 讨论

示例

```
void printCollection(Collection<?> c) { // a wildcard collection
    for (Object o : c) {
        System.out.println(o);
    }
}
```

注意，把 `Collection<Object>` 用作引入参数 `c` 的类型不会有多大用处：该方法只能与具有类型 `Collection<Object>` 的实参一起使用，而这种参数相当少见。作为对比，使用无界通配符允许把任何类型的集合用作参数。

在只对所需要的类型参数具有部分了解的情况下，通配符很有用。

#### 讨论

示例：通配符参数化类型作为数组类型的元素类型。

```
public Method getMethod(Class<?>[] parameterTypes) { ... }
```



可以为通配符指定明确的界限，就像常规的类型变量声明一样。表示上限的语法如下：

```
? extends B
```

其中  $B$  是界限。

### 讨论

示例：有界通配符。

```
boolean addAll(Collection<? extends E> c)
```

这里，该方法是在接口 `Collection<E>` 内声明的，并且设计用于把其进入参数的所有元素添加到一个集合中，该方法就是基于这个集合被调用的。一种自然的倾向是把 `Collection<E>` 用作  $c$  的类型，但这是一种不必要的限制。一种替代方法是把方法自身声明为泛型方法：

```
<T> boolean addAll(Collection<T> c)
```

这个版本足够灵活，但是注意，类型参数在签名中只使用了一次。这反映了如下事实：即类型参数未被用于表达参数类型、返回类型和/或抛出类型之间任何类型的相关性。在缺少这种相关性的情况下，泛型方法被认为是一种糟糕的风格，程序员更喜欢使用通配符。

与方法签名中声明的泛型类型变量不同的是，当使用通配符时，不需要进行类型推理。因此，允许程序员使用以下语法声明较低的通配符界限：

```
? super B
```

其中  $B$  是较低的界限。

### 讨论

示例：较低的通配符界限。

```
Reference(T referent, ReferenceQueue<? super T> queue);
```

这里，可以把引用插入到任何队列中，队列的元素类型是引用的类型  $T$  的超类型。

如果参数是类型变量或通配符，并且两个参数具有不同的类型，则可证实两个类型参数是截然不同的。

### 讨论

通配符与已建立的类型理论的关系是一个有趣的话题，这里我们将简短地间接讨论一下这种关系。

通配符是存在的类型的一种受限形式。假定泛型类型声明  $G<T \text{ extends } B>$ ， $G<?>$  大体类似于某个  $X <: B$ 。  $G<X>$ 。

对更全面的讨论感兴趣的读者应该参阅 Atsushi Igarashi 和 Mirko Viroli 所著的《On

Variance-Based Subtyping for Parametric Types》，它被收集在关于面向对象编程的第16次欧洲大会（ECOOP 2002年）的会议记录中。

通配符的某些细节不同于上述论文中描述的构造，特别是在捕获转换（5.1.10节）的使用方面，而非 Igarashi 和 Viroli 所描述的 `close` 操作。有关通配符的正式说明，参见 Mads Torgersen、Erik Ernst 和 Christian Plesner 在面向对象编程基础（FOOL 2005年）的第12次研讨会上所著的《Wild FJ》。

### 讨论

从历史上讲，通配符是由 Atsushi Igarashi 和 Mirko Viroli 的著作直接演化而来的。该著作本身建立于 Kresten Thorup 和 Mads Torgersen 的早期著作（“Unifying Genericity”，ECOOP 99），以及关于基于声明的变化的长期传统著作[可以追溯到 Pierre America 关于 POOL 的著作（OOPSLA 89）]之上。

#### 4.5.1.1 类型参数包含和等价性

如果在下列规则[其中 $<$ 表示子类型化（4.10节）]之下，可证实  $TA_2$  表示的类型集是  $TA_1$  表示的类型集的子集，则称类型参数  $TA_1$  包含另一个类型参数  $TA_2$ ，记作  $TA_2 \leq TA_1$ ：

- $? \text{ extends } T \leq ? \text{ extends } S \text{ if } T < S$
- $? \text{ super } T \leq ? \text{ super } S \text{ if } S < T$
- $T \leq T$
- $T \leq ? \text{ extends } T$
- $T \leq ? \text{ super } T$

#### 4.5.2 参数化类型的成员和构造函数

设  $C$  是具有形式类型参数  $A_1, \dots, A_n$  的类或接口声明，并且设  $C\langle T_1, \dots, T_n \rangle$  是  $C$  的调用，其中对于  $1 \leq i \leq n$ ， $T_i$  是类型（而不是通配符）。则：

- 设  $m$  是  $C$  中的成员或构造函数声明，如所声明的那样， $C$  的类型是  $T$ 。那么类型  $C\langle T_1, \dots, T_n \rangle$  中的  $m$  的类型（8.2节、8.8.6节）是  $T[A_1 := T_1, \dots, A_n := T_n]$ 。
- 设  $m$  是  $D$  中的成员或构造函数声明，其中  $D$  是一个由  $C$  扩展的类或者由  $C$  实现的接口。设  $D\langle U_1, \dots, U_k \rangle$  是对应于  $D$  的  $C\langle T_1, \dots, T_n \rangle$  的超类型。那么  $C\langle T_1, \dots, T_n \rangle$  中的  $m$  的类型是  $D\langle U_1, \dots, U_k \rangle$  中的  $m$  的类型。

如果相对于参数化类型的任何类型参数是通配符，那么其成员和构造函数的类型是未定义的。

### 讨论

这是没有结果的，因为不可能在不执行捕获转换（5.1.10节）的情况下访问参数化类型的成员，并且不可能在类实例创建表达式中的关键字 `new` 后面使用通配符类型。

## 4.6 类型擦除

类型擦除是指从一些类型（可能包括参数化类型和类型变量）映射到另一些类型（永远不会是参数化类型或类型变量）。我们把类型  $T$  的擦除记为  $|T|$ 。擦除映射定义如下。

- 参数化类型（4.5 节） $G\langle T_1, \dots, T_n \rangle$  的擦除是  $|G|$ 。
- 嵌套类型  $T.C$  的擦除是  $|T|.C$ 。
- 数组类型  $T[]$  的擦除是  $|T|[]$ 。
- 类型变量（4.4 节）的擦除是其最左边界限的擦除。
- 所有其他类型的擦除是类型自身。

方法签名  $s$  的擦除是一个包含与  $s$  同名的签名，以及  $s$  中给定的所有形式参数类型的擦除。

## 4.7 可具体化的类型

由于某些类型信息在编译期间会被擦除，所以并非所有的类型在运行时都可用。在运行时完全可用的类型称为可具体化的类型。当且仅当下列条件之一成立时，一个类型才是可具体化的：

- 它引用一个非泛型类型声明。
- 它是一个参数化类型，其中所有的类型参数都是无界通配符（4.5.1 节）。
- 它是原生类型（4.8 节）。
- 它是基本类型（4.2 节）。
- 它是数组类型（10.1 节），其元素类型是可具体化的。



不使所有泛型类型都可具体化的决定是涉及语言类型系统的最至关重要、最有争议的设计决策之一。

最终，这一决策的最重要的动机是与现有代码兼容。

天真地讲，增加诸如泛型技术（genericity）之类的新构造对于预先存在的代码没有意义。只要用以前的版本编写的每个程序保持其在新版本中的含义，编程语言本质上是与早先的版本兼容的。但是，这个概念（可以称之为语言兼容性）纯粹是理论上的兴趣。真实程序（甚至没有什么价值的程序，如“Hello World”）是由多个编译单元组成的，其中一些编译单元是由 Java 平台提供的（如 `java.lang` 或 `java.util` 的元素）。

那么，实际上平台兼容性的最低要求是：为该平台的以前版本编写的任何程序都可以不加更改地运行在新平台上。

提供平台兼容性的一种方式保持现有平台的功能不变，仅添加新的功能。例如，程序员可以利用泛型技术引入新库，这优于修改 `java.util` 中现有的集合层次结构。

这种模式的缺点是：集合库的预先存在的客户极其难以迁移到新库。集合用于在独立

开发的模块之间交换数据：如果某位供应商决定转变到新型泛型库，那么这位供应商还必须分发其代码的两个版本，以与他们的客户兼容。直至供应商的库被更新为止，才能修改依赖于其他供应商代码的库，以使用泛型技术。如果两个模块互相依赖，则必须同时执行更改。

显然，如上面所概括的，平台兼容性并没有为采纳普适的新特性（如泛型技术）提供现实的途径。因此，泛型类型系统的设计寻求支持迁移兼容性。迁移兼容性允许演进现有的代码以利用泛型技术，而无需在独立开发的软件模块之间强加依赖性。

迁移兼容性的代价是：不可能完全、合理地具体化泛型类型系统，至少在迁移正在发生时如此。

## 4.8 原生类型

为了便于和非泛型遗留代码接口，还可以把某个类型用作参数化类型（4.5 节）的擦除（4.6 节）。这样的类型称为原生类型（raw type）。

更确切地讲，原生类型被定义为：

- 使用的泛型类型声明的名称，而没有任何伴随的实际类型参数。
- 或者
- 原生类型 *R* 的任何非静态类型成员，它不是从 *R* 的超类或超接口派生而来的。

后者的要点自身可能不那么显而易见。下面介绍了一个示例，可供考虑：

```
class Outer<T>{
    T t;
    class Inner {
        T setOuterT(T t1) {t = t1;return t;}
    }
}
```

Inner 成员的类型依赖于 Outer 的类型参数。如果 Outer 是原生的，也必须把 Inner 视作原生的，因为没有对 *T* 的有效绑定。

这条规则仅适用于未被继承的类型成员。由原生类型的超类型将被擦除这条规则可知：依赖于类型变量的被继承的类型成员将会作为原生类型被继承，这将在本节后面进行描述。

上面这些规则的另一个含义是，原生类型的泛型内部类自身只能用作一种原生类型：

```
class Outer<T>{
    class Inner<S> {
        S s;
    }
}
```

不可能作为部分原生类型（一种“稀有”类型）访问 Inner。

```
Outer.Inner<Double> x = null; // illegal
Double d = x.s;
```

由于 Outer 自身是原生类型，因此是它的所有内部类，包括 Inner，因此不可能把任何类型参数传递给它。

仅允许把原生类型用作对遗留代码兼容性的让步。在 Java 编程语言中引入泛型技术之后，在编写的代码中使用原生类型非常让人气馁。Java 编程语言的将来版本有可能禁止使用原生类型。

尝试把参数化类型的类型成员用作原生类型，会引发编译时错误。



这意味着“稀有”类型上的禁令扩展到合格类型被参数化的情况，但是我们尝试把内部类用作原生类型：

```
Outer<Integer>.Inner x = null; // illegal
```

这是与我们上面的讨论相对的情况。这种不成熟的类型没有实用的理由。在遗留代码中，我们应该正确地使用泛型类型，并传递所有需要的实际类型参数。



可以把任何类型的参数实例的值赋予原生类型的变量。

例如，基于子类型化规则（4.10.2 节），把 Vector<String> 赋予 Vector 是可能的。从 Vector 到 Vector<String> 的反方向赋值是不安全的（由于原生向量可能具有不同的元素类型），但是为了支持与遗留代码接口，通过使用未经检查的转换（5.1.9 节）仍然允许这样赋值。在这种情况下，编译器将发出一个未经检查的警告。

原生类型的超类（或者超接口）是其任何参数化调用的超类（超接口）的擦除。

无法从其超类或超接口派生得到的构造函数（8.8 节）的类型、实例方法（8.8 节、9.4 节）或原生类型 C 的非静态字段（8.3 节）M，是其在对应于 C 的泛型声明中的类型的擦除。原生类型 C 的静态成员的类型与其在对应于 C 的泛型声明中的类型相同。

把实际的类型参数传递给从其超类或超接口派生得到的原生类型的非静态类型成员是一个编译时错误。

为了确保始终会标记出潜在违背类型化规则的情况，对原生类型的成员的某些访问会产生警告消息。当访问原生类型的成员或构造函数时，生成警告的规则如下：

- 如果擦除将任何参数的任何类型更改成方法或构造函数，则对原生类型的方法或构造函数的调用会生成一个未经检查的警告。
- 如果擦除更改了原生类型的字段的类型，则对该字段赋值会生成一个未经检查的警告（5.1.9 节）。



对于读取字段或者对于原生类型的类实例创建而言，当参数类型没有改变时，方法调用不需要未经检查的警告（即使结果类型和/或抛出子句发生改变也是如此）。

类的超类型可能是原生类型。对类的成员访问将被视为正常的访问，对超类型的成员访问将被视为对原生类型的访问。在类的构造函数中，调用超类型将被视为原生类型上的方法调用。

示例：原生类型。

```
class Cell<E>
    E value;
    Cell (E v) { value=v; }
    A get() { return value; }
    void set(E v) { value=v; }
}
Cell x = new Cell<String>("abc");
x.value;           // OK, has type Object
x.get();           // OK, has type Object
x.set("def");      // unchecked warning
```

例如：

```
import java.util.*;
class NonGeneric {
    Collection<Number> myNumbers(){return null;}
}
abstract class RawMembers<T> extends NonGeneric implements Collection<String> {
    static Collection<NonGeneric> cng = new ArrayList<NonGeneric>();
    public static void main(String[] args) {
        RawMembers rw = null;
        Collection<Number> cn = rw.myNumbers(); // ok
        Iterator<String> is = rw.iterator(); // unchecked warning
        Collection<NonGeneric> cnn = rw.cng; // ok - static member
    }
}
```

**RawMembers<T>**从 **Collection<String>**超接口继承方法 **Iterator<String>** **iterator()**。但是，类型 **RawMembers** 从其超接口的擦除继承 **iterator()**，这意味着成员 **iterator()** 的返回类型是 **Iterator<<String>** 的擦除，即 **Iterator**。因此，尝试给 **rw.iterator()** 赋值需要从 **Iterator** 到 **Iterator<String>** 的未经检查的转换（5.1.9 节），这会导致发出未经检查的警告。

相反，静态成员 **cng** 会保持其完全参数化类型，甚至当通过原生类型的对象进行访问时也是如此（注意，通过一个实例访问静态成员被认为是一种糟糕的风格，并且会使人气



馁)。成员 `myNumbers` 继承自 `NonGeneric` (它的擦除也是 `NonGeneric`)，因此会保持其完全的参数化类型。

#### 讨论

原生类型与通配符密切相关。它们二者都基于存在的类型。原生类型可以被视作通配符，这类通配符具有故意使之不健全的类型规则，以适应与遗留代码的交互。

从历史上讲，原生类型是在通配符之前出现的；它们最初是在 GJ 中介绍的，并在 Gilad Bracha、Martin Odersky、David Stoutamire 和 Philip Wadler 所著的论文《Making the future safe for the past: Adding Genericity to the Java Programming Language》中进行了描述，该论文收集在 1998 年 10 月的面向对象编程、系统、语言 and 应用程序 (OOPSLA98) 的 ACM 大会的会议录中。

## 4.9 交集类型

交集类型采用  $T_1 \& \dots \& T_n$  ( $n > 0$ ) 的形式，其中  $T_i$  ( $1 \leq i \leq n$ ) 是类型表达式。交集类型出现在捕获转换 (5.1.10 节) 和类型引用 (15.12.2.7 节) 的过程中。直接把交集类型编写为程序的一部分是不可能的；没有语法支持这样做。交集类型的值是一些对象，这些对象是所有的类型  $T_i$  ( $1 \leq i \leq n$ ) 的值。

确定交集类型  $T_1 \& \dots \& T_n$  的成员的方式如下：

- 对于每个  $T_i$  ( $1 \leq i \leq n$ )，假设  $C_i$  是最特定的类或数组类型，满足  $T_i <: C_i$ 。那么对于任何  $i$ ，当  $1 \leq i \leq n$  时，必定有某个  $T_k <: C_k$ ，满足  $C_k <: C_i$ ，否则，就会发生编译时错误。
- 对于  $1 \leq j \leq n$ ，如果  $T_j$  是类型变量，那么设  $IT_j$  是一个接口，其成员与  $T_j$  的公共成员相同；否则，如果  $T_j$  是一个接口，则设  $IT_j$  是  $T_j$ 。
- 这样，交集类型具有与出现交集类型的相同包中声明的具有空主体的类类型 (第 8 章)、直接超类  $C_k$  和直接超接口  $IT_1, \dots, IT_n$  相同的成员。

#### 讨论

交集类型和类型变量的界限之间的区别值得注意。每个类型变量界限都会导致一个交集类型。这种交集类型通常是微不足道的 (即包含单一类型)。

界限的形式是受限制的 (只有第一个元素可以是类或类型变量，只有一个类型变量可以出现在界限中)，以把某些棘手的情形排除在外。但是，捕获转换可能导致创建其界限更泛型的类型变量 (例如，数组类型)。

## 4.10 子类型化

子类型和超类型的关系是类型上的二进制关系。一种类型的超类型是通过直接超类型

关系上的自反和传递闭包获得的, 记作  $S >_1 T$ , 它是通过本节后面给出的规则定义的。我们用  $S :> T$  指示  $S$  和  $T$  之间的超类型关系成立。如果  $S :> T$  并且  $S \neq T$ , 则  $S$  是  $T$  正确的超类型, 记作  $S > T$ 。

如果类型  $T$  的子类型都是类型  $U$  和空类型, 则  $T$  是  $U$  的超类型。我们用  $T <: S$  指示类型  $T$  和  $S$  之间的子类型关系成立。如果  $T <: S$  并且  $S \neq T$ , 则  $T$  是  $S$  正确的子类型, 记作  $T < S$ 。如果  $S >_1 T$ , 则  $T$  是  $S$  的直接子类型, 记作  $T <_1 S$ 。

子类型化不会通过泛型类型进行扩展:  $T <: U$  并不意味着  $C < T > <: C < U >$ 。

#### 4.10.1 基本类型之间的子类型化

下列规则定义了基本类型之间的直接超类型关系:

```
double >_1 float
float >_1 long
long >_1 int
int >_1 char
int >_1 short
short >_1 byte
```

#### 4.10.2 类和接口类型之间的子类型化

设  $C$  是具有零个或多个类型参数 (4.4 节)  $F_1, \dots, F_n$  的类型声明 (4.12.6 节、8.1 节、9.1 节), 这些类型参数具有对应的界限  $B_1, \dots, B_n$ 。这个类型声明定义了一组参数化类型 (4.5 节)  $C_1 < T_1, \dots, T_n >$ , 其中每个参数类型  $T_i$  包括所有类型, 这些类型是对应的界限中列出的所有类型的子类型。也就是说, 对于  $B_i$  中的每个界限类型  $S_i$ ,  $T_i$  是  $S_i[F_1 := T_1, \dots, F_n := T_n]$  的子类型。

给定  $C < F_1, \dots, F_n >$  的类型声明, 参数化类型 (4.5 节)  $C < F_1, \dots, F_n >$  的直接超类型包括以下所有类型:

- $C$  的直接超类。
- $C$  的直接超接口。
- `Object` 类型, 如果  $C$  是不具有直接超接口的接口类型。
- 原生类型  $C$ 。

类型  $C < T_1, \dots, T_n >$  的直接超类型 [其中  $T_i (1 \leq i \leq n)$  是一种类型] 是  $D < U_1 \text{ theta}, \dots, U_k \text{ theta} >$ , 其中:

- $D < U_1, \dots, U_k >$  是  $C < F_1, \dots, F_n >$  的直接超类型,  $\text{theta}$  是代换  $[F_1 := T_1, \dots, F_n := T_n]$ 。
- $C < S_1, \dots, S_n >$ , 当  $1 \leq i \leq n$  时,  $S_i$  包含 (4.5.1 节)  $T_i$ 。

类型  $C < R_1, \dots, R_n >$  的直接超类型 [其中至少有一个  $R_i (1 \leq i \leq n)$  是通配符类型参数] 是  $C < X_1, \dots, X_n >$  的直接超类型, 其中:  $C < X_1, \dots, X_n >$  是对  $C < R_1, \dots, R_n >$  应用捕获转换 (5.1.10 节) 的结果。

交集类型 (4.9 节)  $T_1 \& \dots \& T_n$  的直接超类型是  $T_i$ , 其中  $1 \leq i \leq n$ 。

类型变量 (4.4 节) 的直接超类型是其界限中列出的类型。

空类型的直接超类型是除空类型自身之外的所有引用类型。

除了上述规则之外, 类型变量是其较低界限的直接超类型。

### 4.10.3 数组类型之间的子类型化

下列规则定义了数组类型之间的直接子类型关系:

- 如果  $S$  和  $T$  都是引用类型, 当且仅当  $S >_1 T$  时,  $S[] >_1 T[]$ 。
- `Object`  $>_1$  `Object[]`
- `Cloneable`  $>_1$  `Object[]`
- `java.io.Serializable`  $>_1$  `Object[]`
- 如果  $p$  是基本类型, 则:
  - ◆ `Object`  $>_1$   $p[]$
  - ◆ `Cloneable`  $>_1$   $p[]$
  - ◆ `java.io.Serializable`  $>_1$   $p[]$

## 4.11 在何处使用类型

当类型出现在声明中或者某些表达式中时, 就会使用到类型。

下列代码段包含大多数类型使用方法的一个或多个实例:

```
import java.util.Random;
class MiscMath<T extends Number>{
    int divisor;
    MiscMath(int divisor) {
        this.divisor = divisor;
    }
    float ratio(long l) {
        try {
            l /= divisor;
        } catch (Exception e) {
            if (e instanceof ArithmeticException)
                l = Long.MAX_VALUE;
            else
                l = 0;
        }
        return (float)l;
    }
    double gausser() {
        Random r = new Random();
        double[] val = new double[2];
        val[0] = r.nextGaussian();
        val[1] = r.nextGaussian();
    }
}
```

```

        return (val[0] + val[1]) / 2;
    }
    Collection<Number> fromArray(Number[] na) {
        Collection<Number> cn = new ArrayList<Number>();
        for (Number n : na) {
            cn.add(n);
        }
        return cn;
    }
    void <S> loop(S s){ this.<S>loop(s);}
}

```

在这个示例中，类型用在下列声明中：

- 导入的类型（7.5 节）：这里声明了类型 `Random`，它是从 `java.util` 包的类型 `java.util.Random` 中导入的。
- 字段，包括类变量和类的实例变量（8.3 节），以及接口的常量（9.3 节）：在这里，类 `MiscMath` 中的字段 `divisor` 被声明为类型 `int`。
- 方法参数（8.4.1 节）：在这里，方法 `ratio` 的参数 `l` 被声明为类型 `long`。
- 方法结果（8.4 节）：在这里，方法 `ratio` 的结果被声明为类型 `float`，方法 `gausser` 的结果被声明为类型 `double`。
- 构造函数参数（8.8.1 节）：在这里，`MiscMath` 的构造函数的参数被声明为类型 `int`。
- 局部变量（14.4 节、14.14 节）：方法 `gausser` 的局部变量 `r` 和 `val` 被声明为类型 `Random` 和 `double[]`（`double` 数组）。
- 异常处理程序参数（14.20 节）：在这里，`catch` 子句的异常处理程序参数 `e` 被声明为类型 `Exception`。
- 类型变量（4.4 节）：在这里，类型变量 `T` 把 `Number` 作为其声明的界限。

而在以下类型的表达式中：

- 类实例创建（15.9 节）：在这里，通过使用类型 `Random` 的类实例创建表达式初始化方法 `gausser` 的局部变量 `r`。
- 泛型类（8.12 节）实例创建（15.9 节）：这里，在表达式 `new ArrayList<Number>()` 中把 `Number` 用作类型参数。
- 数组创建（15.10 节）：在这里，通过创建一个大小为 2 的 `double` 数组的数组创建表达式，初始化方法 `gausser` 的局部变量 `val`。
- 泛型方法（8.4.4 节）或构造函数（8.8.4 节）调用（15.12 节）：在这里，方法 `loop` 通过显式类型参数 `S` 调用它自身。
- 强制转换（15.16 节）：在这里，方法 `ratio` 的返回语句在强制转换中使用 `float` 类型。
- `instanceof` 运算符（15.20.2 节）：在这里，`instanceof` 运算符测试 `e` 是否是类型 `ArithmeticException` 兼容的赋值。
- 类型还用作参数化类型的参数：在这里，类型 `Number` 用作参数化类型 `Collection<Number>` 中的一个参数。

## 4.12 变量

变量是一个存储位置，并且具有关联的类型，有时称之为编译时类型，它可以是基本类型（4.2 节）或引用类型（4.3 节）。通过赋值或者前缀运算符++（递增）或后缀运算符--（递减）（15.14.2 节、15.14.3 节、15.15.1 节、15.15.2 节），可以更改变量的值。

变量的值与其类型的兼容性是通过 Java 编程语言的设计保证的，只要程序不引发未经检查的警告（4.12.1 节）即可。默认值是兼容的（4.12.5 节），并且对变量的所有赋值都会检查赋值兼容性（5.2 节），这通常是在编译时进行的，但是在单独一种涉及数组的情况下，是在运行时执行检查的（10.10 节）。

### 4.12.1 基本类型的变量

基本类型的变量总是存储严格的基本类型的值。

### 4.12.2 引用类型的变量

类类型  $T$  的变量可以存储空引用，或者指向类  $T$  或其任何子类的实例的引用。接口类型的变量可以存储空引用，或者指向实现该接口的任何类的任何实例的引用。

#### 讨论

注意，不保证变量总是引用其声明类型的子类型，而只保证引用声明类型的子类或子接口。这是由于下面将讨论的堆污染的可能性。

如果  $T$  是基本类型，那么“ $T$  的数组”这种类型的变量可以存储空引用，或者指向“ $T$  的数组”这种类型的任何数组的引用；如果  $T$  是引用类型，那么“ $T$  的数组”这种类型的变量可以存储空引用，或者指向“ $S$  的数组”这种类型的任何数组，其中类型  $S$  是类型  $T$  的子类或子接口。此外，`Object[]` 类型的变量可以存储任何引用类型的数组。`Object` 类型的变量可以存储空引用，或者指向任何对象的引用，无论该对象是类实例还是数组。

#### 4.12.2.1 堆污染

参数化类型的变量有可能引用一个不是那种参数化类型的对象，这种情况称为堆污染。仅当程序执行一些将会在编译时引发未经检查的警告时，才会发生这种情况。

#### 讨论

例如，下列代码：

```
List l = new ArrayList<Number>();  
List<String> ls = l; // unchecked warning
```

将会引发一个未经检查的警告，由于它不能确定在编译时（在编译时类型检查规则的限制内）或者在运行时，变量 `l` 是否确实引用 `List<String>`。



如果执行上面的代码，就会发生堆污染，因为声明为 `List<String>` 的变量 `ls` 引用的是一个实际上不是 `List<String>` 的值。

这个问题无法在运算时确定，这是由于类型变量未具体化，因此在运行时实例不会携带关于用于创建它们的实际类型参数的任何信息。

在上面给出的简单示例中，看起来在编译时确定情形并给出编译错误应该是直观的。但是在通常（和典型）情况下，变量 `l` 的值可能是调用独立编译的方法的结果，或者它的值可能依赖于任意的控制流程。

因此，上面的代码极其反常，确实是一种非常糟糕的风格。

仅当结合未利用参数化类型的遗留代码和利用参数化类型的更多现代代码时，才应该使用把原生类型的值赋予参数化类型的变量这种赋值方式。

如果未执行需要未经检查警告的运算，就不会发生堆污染。注意，这并不意味着仅当未经检查警告实际发生时才会发生堆污染。有可能运行一个程序，其中有些二进制位被用于 Java 编程语言的旧版本的编译器编译，或者被允许禁止未经检查的警告的编译器使用。这种实践至多是反常的。

相反，有可能出现的一种情况是：尽管执行可能（也许确实会）引发未经检查的警告的代码，也不会发生堆污染。的确，良好的编程实践需要程序员对于任何未经检查的警告都要使他们自己感到满意，并且需要代码是正确的，同时不会发生堆污染。

---

变量将总是引用一个对象，它是实现参数化类型的类的一个实例。

---

例如，上述示例中 `l` 的值总是 `List`。

---

### 4.12.3 变量的种类

变量共有 7 种：

(1) 类变量是在类声明中使用关键字 `static` 声明的字段（8.3.1.1 节），或者在接口声明中使用或不使用关键字 `static` 声明的字段（9.3 节）。当准备好类或接口（12.3.2 节）时，就会创建类变量并将其初始化为一个默认值（4.12.5 节）。当卸载类或接口（12.7 节）时，就会有效地使类变量停止存在。

(2) 实例变量是在类声明中不使用关键字 `static` 声明的字段（8.3.1.1 节）。如果类 `T` 有一个字段 `a`，它是一个实例变量，那么就会创建一个新的实例变量 `a`，并将其初始化为一个默认值（4.12.5 节），作为类 `T` 或其任何子类的每个新创建的对象的一部分（8.1.4 节）。在完成对象的任何必要的终结（12.6 节）之后，当作为字段的对象不再被引用时，就会有效地使实例变量停止存在。

(3) 数组元素是未命名的变量，无论何时创建一个作为数组的新对象（15.10 节），都会创建这些变量并将其初始化为默认值（4.12.5 节）。当数组不再被引用时，就会有效地使



数组元素停止存在。参见第 10 章以了解数组的描述。

(4) 方法参数 (8.4.1 节) 指定传递给方法的参数值。对于方法声明中声明的每个参数, 每当调用该方法 (15.12 节) 时, 都会创建一个新的参数变量。新变量会来自方法调用的相应参数值进行初始化。当方法体执行完成时, 就会有效地使方法参数停止存在。

(5) 构造函数参数 (8.8.1 节) 指定传递给构造函数的参数值。对于构造函数声明中声明的每个参数, 每当类实例创建表达式 (15.9 节) 或显式构造函数调用 (8.8.7 节) 调用该构造函数时, 就会创建一个新的参数变量。新变量会来自创建表达式或构造函数调用的相应参数值进行初始化。当构造函数体执行完成时, 就会有效地使构造函数参数停止存在。

(6) 每当 `catch` 子句或 `try` 语句捕获一个异常时, 就会创建一个异常处理程序参数 (14.20 节)。新变量会用与异常关联的实际对象进行初始化 (11.3 节、14.18 节)。当与 `catch` 子句关联的代码块执行完成时, 就会有效地使异常处理程序参数停止存在。

(7) 局部变量是通过局部变量声明语句 (14.4 节) 声明的。无论何时控制流程进入代码块 (14.2 节) 或 `for` 语句 (14.14 节), 都会为代码块或 `for` 语句内紧接着包含的局部变量声明语句中声明的每个局部变量创建一个新变量。局部变量声明语句可以包含初始化变量的表达式。但是, 直到执行声明局部变量的局部变量声明语句时, 才会用初始化表达式初始化局部变量 [在局部变量被初始化或者为其赋值之前, 明确赋值 (第 16 章) 的规则会阻止使用局部变量的值]。当代码块或 `for` 语句执行完成时, 就会有效地使局部变量停止存在。

如果不是用于一种异常情况, 则总是可以把局部变量看作是在执行其局部变量声明语句时创建的。异常情况涉及 `switch` 语句 (14.11 节), 它有可能用于控制进入代码块, 但是会绕过局部变量声明语句的执行。然而, 由于明确赋值 (第 16 章) 规则强加的限制, 在通过赋值表达式 (15.26 节) 明确为局部变量赋值之前, 将不能使用这样一种绕过局部变量声明语句的方式来声明局部变量。

下面的示例包含多种不同类型的变量:

```
class Point {
    static int numPoints;    // numPoints is a class variable
    int x, y;                // x and y are instance variables
    int[] w = new int[10];   // w[0] is an array component
    int setX(int x) {        // x is a method parameter
        int oldx = this.x;   // oldx is a local variable
        this.x = x;
        return oldx;
    }
}
```

#### 4.12.4 final 变量

变量可以被声明为 `final`。`final` 变量可能只被赋值一次。除非在赋值之前立即明确地取消对 `final` 变量进行赋值, 否则, 如果对 `final` 变量赋值, 将会引发编译时错误。

*blank final* 是其声明缺少初始化语句的 `final` 变量。

一旦为 `final` 变量赋值, 则它总是包含相同的值。如果 `final` 变量存储一个指向对象的

引用，那么就可以通过该对象上执行的操作来更改对象的状态，但是变量将总是引用相同的对象。这也适用于数组，这是因为数组是对象：如果 `final` 变量存储一个指向数组的引用，那么就可以通过该数组上执行的操作来更改数组的元素，但是变量将总是引用相同的数组。

将变量声明为 `final` 可以用作有用的文件证明，即其值将不会改变，并且可以帮助避免编程错误。

在下面的示例中：

```
class Point {
    int x, y;
    int useCount;
    Point(int x, int y) { this.x = x; this.y = y; }
    final static Point origin = new Point(0, 0);
}
```

类 `Point` 声明了一个 `final` 类变量 `origin`。`origin` 变量存储一个指向对象的引用，该对象是类 `Point` 的实例，它的坐标为 (0, 0)。变量 `Point.origin` 的值永远不会变化，因此它总是引用相同的 `Point` 对象，该对象是通过其初始化语句创建的。但是，`Point` 对象上的操作可能改变其状态——例如，修改它的 `useCount`，或者甚至误导性地更改其 `x` 或 `y` 坐标。

我们调用一个基本类型或 `String` 类型的变量，该变量是 `final` 变量，并用编译时常量表达式（15.28 节）将其初始化为一个常量变量。无论一个变量是否是常量变量，它都具有关于类初始化（12.4.1 节）、二进制兼容性（13.1 节、13.4.9 节）和明确赋值（第 16 章）的含义。

#### 4.12.5 变量的初始值

在使用变量的值之前，程序中的每个变量都必须有一个值：

- 每个类变量、实例变量或数组元素都会在创建时用默认值对其进行初始化（15.9 节、15.10 节）：
  - ◆ 对于类型 `byte`，默认值是 0，即 `(byte)0` 的值。
  - ◆ 对于类型 `short`，默认值是 0，即 `(short)0` 的值。
  - ◆ 对于类型 `int`，默认值是零，即 0。
  - ◆ 对于类型 `long`，默认值是 0，即 `0L`。
  - ◆ 对于类型 `float`，默认值是正 0，即 `0.0f`。
  - ◆ 对于类型 `double`，默认值是正 0，即 `0.0d`。
  - ◆ 对于类型 `char`，默认值是空字符，即 `'\u0000'`。
  - ◆ 对于类型 `boolean`，默认值是 `false`。
  - ◆ 对于所有引用类型（4.3 节），默认值是 `null`。
- 每个方法参数（8.4.1 节）都会被初始化为方法调用者提供的对应参数值（15.12 节）。
- 每个构造函数参数（8.8.1 节）都会被初始化为类实例创建表达式（15.9 节）或显式构造函数调用（8.8.7 节）提供的对应参数值。
- 异常处理程序参数（14.20 节）将被初始化为抛出一个表示异常的对象（11.3 节、14.18

节)。

- 在使用局部变量(14.4节、14.14节)之前,必须通过初始化(14.4节)或赋值(15.26节)显式地为其赋值,编译器可以使用明确赋值(第16章)的规则对这种赋值方式进行验证。

示例程序:

```
class Point {
    static int npoints;
    int x, y;
    Point root;
}

class Test {
    public static void main(String[] args) {
        System.out.println("npoints=" + Point.npoints);
        Point p = new Point();
        System.out.println("p.x=" + p.x + ", p.y=" + p.y);
        System.out.println("p.root=" + p.root);
    }
}
```

输出如下:

```
npoints=0
p.x=0, p.y=0
p.root=null
```

这个示例程序阐释了 npoints 的默认初始化,当准备 Point 类(12.3.2节)时,它就会发生;还阐释了 x、y 和 root 的默认初始化,当实例化一个新的 Point 时,就会发生。参见第12章,以了解关于加载、链接和初始化类及接口的完整描述,以及实例化类以生成新的类实例的描述。

#### 4.12.6 类型、类和接口

在 Java 编程语言中,每个变量和每个表达式都有一个可以在编译时确定的类型。这个类型可以是基本类型或引用类型。引用类型包括类类型和接口类型。引用类型是通过类型声明引入的,包括类声明(8.1节)和接口声明(9.1节)。我们通常使用术语“类型”来指示类或接口。

每个对象都属于某个特殊的类:创建表达式中提及的用于生成对象的类、其 Class 对象用于调用自反方法以生成对象的类,或者用于通过字符串串接运算符+ (15.18.1节)隐式创建对象的 String 对象。这个类被称为对象的类(数组也是一个类,本节末尾描述了这一点)。对象被称为其类或这个类的所有超类的实例。

有时称一个变量或表达式具有“运行时类型”。这是指在运行时被变量或表达式的值引用的对象的类,假定该值不是 null。

总是会声明变量的编译时类型,并且可以在编译时推导出表达式的编译时类型。编译

时类型限制了在运行时变量可以存储或者表达式可以产生的可能值。如果运行时的值是一个不为 null 的引用，则它引用的是一个具有类的对象或数组，并且这个类必须与编译时类型兼容。

即使变量或表达式可能具有一个接口类型的编译时类型，也不会有接口的实例存在。其类型是接口类型的变量或表达式可以引用其类实现了（8.1.5 节）该接口的任何对象。

下面的示例阐释了创建新对象以及变量的类型和对象的类之间的区别：

```
public interface Colorable {
    void setColor(byte r, byte g, byte b);
}
class Point { int x, y; }
class ColoredPoint extends Point implements Colorable {
    byte r, g, b;
    public void setColor(byte rv, byte gv, byte bv) {
        r = rv; g = gv; b = bv;
    }
}
class Test {
    public static void main(String[] args) {
        Point p = new Point();
        ColoredPoint cp = new ColoredPoint();
        p = cp;
        Colorable c = cp;
    }
}
```

在这个示例中：

- 类 Test 的 main 方法的局部变量 p 具有类型 Point，并在初始时被赋予一个指向类 Point 的新实例的引用。
- 局部变量 cp 同样把 ColoredPoint 作为其类型，并在初始时被赋予一个指向类 ColoredPoint 的新实例的引用。
- 将 cp 赋值给变量 p，导致 p 存储一个指向 ColoredPoint 对象的引用。这是允许的，因为 ColoredPoint 是 Point 的子类，因此类 ColoredPoint 与类型 Point 是赋值兼容的（5.2 节）。ColoredPoint 对象包括对 Point 的所有方法的支持。除了其特殊的字段 r、g 和 b 外，它还具有类 Point 的字段，即 x 和 y。
- 局部变量 c 把接口类型 Colorable 作为其类型，因此它可以存储一个指向其类实现了 Colorable 的任何对象的引用，确切地讲，它可以存储一个指向 ColoredPoint 的引用。

---

注意：诸如 new Colorable() 之类的表达式无效，因为不可能创建接口的实例，而只能创建类的实例。

---

每个数组也有一个类：当为数组对象调用方法 `getClass` 时，该方法将返回一个类对象（类 `Class` 的对象），它表示数组的类。

数组的类具有奇特的名称，它们不是有效的标识符：例如，`int` 元素的数组的类具有名称 “[I”，因此表达式：

```
new int[10].getClass().getName()
```

的值是字符串 “[I”；参见 `Class.getName` 的规范以了解详细信息。

我常常悠闲地坐在松软的草皮上，此时，万事皆抛云外。

——William Wordsworth, 《To the Same Flower》

## 转换和提升

艺术不存在于这样的地方，这里人们只看到晋职，而没有流汗。  
——威廉·莎士比亚，《皆大欢喜》第二幕第三场

用 Java 编程语言编写的每个表达式都有一个类型，可以通过表达式的结构以及表达式中提及的值、变量和方法的类型推导出这个类型。但是，有可能在表达式的类型不合适的环境中编写一个表达式。在某些情况下，这可能导致编译时错误。在其他情况下，环境也许可以接受与表达式的类型相关的类型；为方便起见，Java 编程语言不需要程序员显式地指定类型转换，而是执行从表达式的类型到其周围环境可接受的类型的隐式转换。

从类型 *S* 到类型 *T* 的特定转换允许在编译时把类型 *S* 的表达式看作好像它具有类型 *T* 一样。在某些情况下，这需要在运行时采取相应的动作，以检查转换的有效性，或者把表达式的运行时的值转变成一种适合于新类型 *T* 的形式。例如：

- 从类型 `Object` 到类型 `Thread` 的转换需要执行运行时检查，以确保运行时的值实际上就是类 `Thread` 或其子类之一的实例；如果不是这样，则会抛出一个异常。
- 从类型 `Thread` 到类型 `Object` 的转换不需要采取运行时动作；`Thread` 是 `Object` 的子类，因此，类型 `Thread` 的表达式产生的任何引用都是类型 `Object` 的有效引用值。
- 从类型 `int` 到类型 `long` 的转换需要执行从 32 位整型值到 64 位 `long` 表示的运行时符号扩展。这不会丢失信息。

从类型 `double` 到类型 `long` 的转换需要执行从 64 位浮点值到 64 位整型表示的重要转换。依赖于实际的运行时值，有可能会丢失信息。

在每种转换环境中，只允许进行某些特定的转换。为了方便描述，在 Java 编程语言中，把可能进行的特定转换分组为几大类：

- 同一性转换
- 扩展基本转换
- 收缩基本转换
- 扩展引用转换
- 收缩引用转换
- 装箱转换



- 拆箱转换
- 未经检查的转换
- 捕获转换
- 字符串转换
- 值集转换

可能在其中发生表达式转换的环境有 5 种。每种环境都允许上面指定的某些类别中的转换，而不允许另外一些类别中的转换。术语“转换”还用于描述为这种环境选择特定转换的过程。例如，我们把作为方法调用中的实参的表达式称为服从“方法调用转换”，这意味着依据方法调用参数环境的规则，将会为那个表达式隐式选择一种特定的转换。

一种转换环境是数值运算符（如+或\*）的操作数。这种操作数的转换过程称为数值提升。提升的特殊性在于：就二元运算符而言，为一个操作数选择的转换可能部分依赖于另一个操作数表达式的类型。

本章首先描述了转换的 11 个类别（5.1 节），包括字符串串接运算符+允许的到 String 的特殊转换。然后描述了 5 种转换环境：

- 赋值转换（5.2 节、15.26 节）把表达式的类型转换成指定变量的类型。赋值转换可能导致在运行时抛出 `OutOfMemoryError` [作为装箱转换（5.1.7 节）的结果]、`NullPointerException` [作为拆箱转换（5.1.8 节）的结果] 或者 `ClassCastException` [作为未经检查的转换（5.1.9 节）的结果]。
- 方法调用转换（5.3 节、15.9 节、15.12 节）适用于方法或构造函数调用中的每个参数，只有一种情况除外，即方法调用转换执行与赋值转换相同的转换。方法调用转换可能导致在运行时抛出 `OutOfMemoryError` [作为装箱转换（5.1.7 节）的结果]、`NullPointerException` [作为拆箱转换（5.1.8 节）的结果] 或者 `ClassCastException` [作为未经检查的转换（5.1.9 节）的结果]。
- 强制转换（5.5 节）将表达式的类型转换成通过强制转换运算符显式指定的类型（15.16 节）。与赋值或方法调用转换相比，它更具包容性，允许除字符串转换之外的任何特定的转换，但是从某些类型强制转换到引用类型可能会在运行时引发异常。
- 字符串转换（5.4 节、15.18.1 节）允许把任何类型转换成 String 类型。
- 数值提升（5.6 节）把数值运算符的操作数提升到泛型类型，以便可以执行运算。

下面是多种转换环境的一些示例：

```
class Test {
    public static void main(String[] args) {
        // Casting conversion (§5.4) of a float literal to
        // type int. Without the cast operator, this would
        // be a compile-time error, because this is a
        // narrowing conversion (§5.1.3):
        int i = (int)12.5f;
        // String conversion (§5.4) of i's int value:
        System.out.println("(int)12.5f==" + i);
        // Assignment conversion (§5.2) of i's value to type
        // float. This is a widening conversion (§5.1.2):
```

```

float f = i;
// String conversion of f's float value:
System.out.println("after float widening: " + f);
// Numeric promotion (§5.6) of i's value to type
// float. This is a binary numeric promotion.
// After promotion, the operation is float*float:
System.out.print(f);
f = f * i;
// Two string conversions of i and f:
System.out.println("** + i + "==" + f);
// Method invocation conversion (§5.3) of f's value
// to type double, needed because the method Math.sin
// accepts only a double argument:
double d = Math.sin(f);
// Two string conversions of f and d:
System.out.println("Math.sin(" + f + ")==" + d);
}
}

```

输出结果如下：

```

(int)12.5f==12
after float widening: 12.0
12.0*12==144.0
Math.sin(144.0)==-0.49102159389846934

```

## 5.1 转换的种类

Java 编程语言中的特定类型转换分为以下类别。

### 5.1.1 同一性转换

对于任何类型而言，均允许从一种类型到那种相同类型的转换。

这似乎没有多大意义，但是，它有两个实际的后果。首先，如果只有平常的同一性转换，那么总是允许一个转换开始于具有希望的类型，从而允许每个表达式都服从转换的简单规定的规则。其次它意味着，出于清晰的目的，允许程序包括冗余的强制转换运算符。

### 5.1.2 扩展基本转换

下列 19 种基本类型上的特定转换称为扩展基本转换：

- byte 到 short、int、long、float 或 double
- short 到 int、long、float 或 double
- char 到 int、long、float 或 double
- int 到 long、float 或 double
- long 到 float 或 double

- float 到 double

扩展基本转换不会丢失关于数值总量的信息。实际上，从整型扩展到另一种整型的转换根本不会丢失任何信息；数值会得到准确的保持。在 `strictfp` 表达式中从 `float` 扩展到 `double` 的转换也会准确地保持数值；但是，对于不是 `strictfp` 的转换，则可能会丢失关于被转换值的总量的信息。

从 `int` 或 `long` 值到 `float`，或者从 `long` 值到 `double` 的转换都可能导致精度损失——也就是说，结果可能丢失值的一些最低有效位。在这种情况下，得到的浮点值将会是使用 IEEE 754 四舍五入至最接近的值模式（4.2.4 节）产生的整型值正确四舍五入的版本。

从带符号整型值到整型 `T` 的扩展转换，仅仅只是对整型值的 2 的补码表示进行符号扩展，以填充扩展的格式。从 `char` 到整型 `T` 的扩展转换将会对该 `char` 值的表示进行零扩展，以填充扩展的格式。

尽管可能会发生精度损失这一事实，基本类型之间的扩展转换永远不会导致运行时异常（第 11 章）。

下面是一个损失精度的扩展转换的示例：

```
class Test {
    public static void main(String[] args) {
        int big = 1234567890;
        float approx = big;
        System.out.println(big - (int)approx);
    }
}
```

输出如下：

-46

因此指示从类型 `int` 到类型 `float` 的转换过程中会丢失信息，这是因为 `float` 类型的值不能精确到 9 个有效位。

### 5.1.3 收缩基本转换

下列 22 种基本类型上的特定转换称为收缩基本转换：

- `short` 到 `byte` 或 `char`
- `char` 到 `byte` 或 `short`
- `int` 到 `byte`、`short` 或 `char`
- `long` 到 `byte`、`short`、`char` 或 `int`
- `float` 到 `byte`、`short`、`char`、`int` 或 `long`
- `double` 到 `byte`、`short`、`char`、`int`、`long` 或 `float`

收缩转换可能会丢失关于数值总量的信息，并且还可能会损失精度。

从带符号整数到整型 `T` 的收缩转换只是简单地丢弃除  $n$  个最低阶位以外的其他所有位，其中  $n$  是用于表示类型 `T` 的位的个数。除了可能丢失关于数值数量的信息外，这还可能导致结果值的符号不同于输入值的符号。

从 char 到整型  $T$  的收缩转换同样只是简单地丢弃除  $n$  个最低阶位以外的其他所有位, 其中  $n$  是用于表示类型  $T$  的位的个数。除了可能丢失关于数值数量的信息外, 这还可能导致结果值为负数, 即使 char 表示的是 16 位无符号整型值。

从浮点数到整型  $T$  的收缩转换采用两个步骤:

(1) 在第一步, 如果  $T$  是 long, 则将浮点数转换成 long; 如果  $T$  是 byte、short、char 或 int, 则将浮点数转换成 int, 如下:

- ◆ 如果浮点数是 NaN (4.2.3 节), 则该转换的第一步的结果是 int 或 long 0。
- ◆ 否则, 如果浮点数不是无穷大, 则将浮点值四舍五入为一个整型值  $V$ , 使用 IEEE 754 四舍五入到 0 模式 (4.2.3 节) 来四舍五入到 0。这有两种情况:
  - ◇ 如果  $T$  是 long, 并且该整型值可以表示为 long, 则第一步的结果是 long 值  $V$ 。
  - ◇ 否则, 如果整型值可以表示为 int, 则第一步的结果是 int 值  $V$ 。
- ◆ 否则, 下列两种情况之一必定成立:
  - ◇ 该值必定极小 (大数的负值或负无穷大), 并且第一步的结果是 int 或 long 类型可表示的最小值。
  - ◇ 该值必定极大 (大数的正值或正无穷大), 并且第一步的结果是 int 或 long 类型可表示的最大值。

(2) 在第二步中:

- ◆ 如果  $T$  是 int 或 long, 则转换的结果是第一步的结果。
- ◆ 如果  $T$  是 byte、char 或 short, 则转换的结果是收缩转换到第一步的结果的类型  $T$  (5.1.3 节) 的结果。

示例:

```
class Test {
    public static void main(String[] args) {
        float fmin = Float.NEGATIVE_INFINITY;
        float fmax = Float.POSITIVE_INFINITY;
        System.out.println("long: " + (long)fmin + ".." + (long)fmax);
        System.out.println("int: " + (int)fmin + ".." + (int)fmax);
        System.out.println("short: " + (short)fmin + ".." + (short)fmax);
        System.out.println("char: " + (int)(char)fmin + ".." + (int)(char)fmax);
        System.out.println("byte: " + (byte)fmin + ".." + (byte)fmax);
    }
}
```

输出结果如下:

```
long: -9223372036854775808..9223372036854775807
int: -2147483648..2147483647
short: 0..-1
char: 0..65535
byte: 0..-1
```

char、int 和 long 的结果并不令人感到奇怪, 它们都产生该类型的最小和最大可表示值。

byte 和 short 的结果会丢失关于数值的符号和数量的信息, 并且还会损失精度。通

过检查最小和最大 `int` 的低阶位，可以理解这些结果。用十六进制表示，最小的 `int` 是 `0x80000000`，最大的 `int` 是 `0x7fffffff`。本示例解释了 `short` 结果，它们是这些值的低 16 位，即 `0x0000` 和 `0xffff`；还解释了 `char` 结果，它们也是这些值的低 16 位，即 `'\u0000'` 和 `'\uffff'`；同时解释了 `byte` 结果，它们是这些值的低 8 位，即 `0x00` 和 `0xff`。

尽管可能发生上溢、下溢或其他的信息丢失，基本类型之间的收缩转换永远不会导致运行时异常（第 11 章）。

下面是一个小测试程序，它演示了许多丢失信息的收缩转换：

```
class Test {
    public static void main(String[] args) {
        // A narrowing of int to short loses high bits:
        System.out.println("(short)0x12345678==0x" +
            Integer.toHexString((short)0x12345678));
        // A int value not fitting in byte changes sign and magnitude:
        System.out.println("(byte)255==" + (byte)255);
        // A float value too big to fit gives largest int value:
        System.out.println("(int)1e20f==" + (int)1e20f);
        // A NaN converted to int yields zero:
        System.out.println("(int)NaN==" + (int)Float.NaN);
        // A double value too large for float yields infinity:
        System.out.println("(float)-1e100==" + (float)-1e100);
        // A double value too small for float underflows to zero:
        System.out.println("(float)1e-50==" + (float)1e-50);
    }
}
```

本测试程序产生以下输入：

```
(short)0x12345678==0x5678
(byte)255==-1
(int)1e20f==2147483647
(int)NaN==0
(float)-1e100==-Infinity
(float)1e-50==0.0
```

#### 5.1.4 扩展基本转换和收缩基本转换

下面的转换结合了扩展基本转换和收缩基本转换：

- `byte` 到 `char`

首先，通过扩展基本转换将 `byte` 转换成 `int`，然后通过收缩基本转换把得到的 `int` 转换成 `char`。

#### 5.1.5 扩展引用转换

假定 `S` 是 `T` 的子类型（4.10 节），则从任何类型 `S` 到任何类型 `T` 都存在扩展引用转换。扩展引用转换从不需要在运行时采取特殊的动作，因而从来不会在运行时抛出异常。

扩展引用转换只存在于这样的情况下，即当把一个引用看成以某种可以在编译时证实为正确的方式具有一些其他类型时。

分别参见第8章、第9章和第10章，以了解有关类、接口和数组的详细规范。

### 5.1.6 收缩引用转换

下列转换称为收缩引用转换：

- 从任何引用类型 *S* 到任何引用类型 *T*，假定 *S* 是 *T* 的合适的超类型（4.10 节）。（一个重要的特例是，从类类型 `Object` 到任何其他引用类型都有一个收缩转换。）
- 从任何类类型 *C* 到任何非参数化接口类型 *K*，假定 *C* 不是 `final` 类型并且不需要实现 *K*。
- 从任何接口类型 *J* 到任何不是 `final` 的非参数化类类型 *C*。
- 从接口类型 `Cloneable` 和 `java.io.Serializable` 到任何数组类型 *T*[]。
- 从任何接口类型 *J* 到任何非参数化接口类型 *K*，假定 *J* 不是 *K* 的子接口。
- 从任何数组类型 *SC*[] 到任何数组类型 *TC*[]，假定 *SC* 和 *TC* 都是引用类型，并且从 *SC* 到 *TC* 有一个收缩转换。

这些转换需要在运行时执行测试，以查明实际的引用值是否是新类型的合法值。如果不是，则会抛出 `ClassCastException`。

### 5.1.7 装箱转换

装箱转换把基本类型的值转换成相应的引用类型的值。确切地讲，下列 8 种转换称为装箱转换：

- 从类型 `boolean` 到类型 `Boolean`
- 从类型 `byte` 到类型 `Byte`
- 从类型 `char` 到类型 `Character`
- 从类型 `short` 到类型 `Short`
- 从类型 `int` 到类型 `Integer`
- 从类型 `long` 到类型 `Long`
- 从类型 `float` 到类型 `Float`
- 从类型 `double` 到类型 `Double`

在运行时，装箱转换的工作方式如下：

- 如果 *p* 是 `boolean` 类型的值，那么装箱转换将把 *p* 转换成类和类型 `Boolean` 的引用 *r*，满足 `r.booleanValue() == p`
- 如果 *p* 是 `byte` 类型的值，那么装箱转换将把 *p* 转换成类和类型 `Byte` 的引用 *r*，满足 `r.byteValue() == p`
- 如果 *p* 是 `char` 类型的值，那么装箱转换将把 *p* 转换成类和类型 `Character` 的引用 *r*，满足 `r.charValue() == p`
- 如果 *p* 是 `short` 类型的值，那么装箱转换将把 *p* 转换成类和类型 `Short` 的引用 *r*，



满足 `r.shortValue() == p`

- 如果 `p` 是 `int` 类型的值, 那么装箱转换将把 `p` 转换成类和类型 `Integer` 的引用 `r`, 满足 `r.intValue() == p`
- 如果 `p` 是 `long` 类型的值, 那么装箱转换将把 `p` 转换成类和类型 `Long` 的引用 `r`, 满足 `r.longValue() == p`
- 如果 `p` 是 `float` 类型的值, 那么:
  - ◆ 如果 `p` 不是 `NaN`, 则装箱转换将把 `p` 转换成类和类型 `Float` 的引用 `r`, 满足 `r.floatValue()` 求值结果为 `p`
  - ◆ 否则, 装箱转换将把 `p` 转换成类和类型 `Float` 的引用 `r`, 满足 `r.isNaN()` 求值结果为真
- 如果 `p` 是 `double` 类型的值, 那么:
  - ◆ 如果 `p` 不是 `NaN`, 则装箱转换将把 `p` 转换成类和类型 `Double` 的引用 `r`, 满足 `r.doubleValue()` 求值结果为 `p`
  - ◆ 否则, 装箱转换将把 `p` 转换成类和类型 `Double` 的引用 `r`, 满足 `r.isNaN()` 求值结果为真。
- 如果 `p` 是任何其他类型的值, 那么装箱转换就等价于同一性转换 (5.1.1 节)。

如果被装箱的 `p` 值是 `true`、`false`, 一个 `byte`、一个在 `\u0000~\u007f` 之间的 `char`, 或者一个在 `-128~127` 之间的 `int` 或 `short` 数字, 设 `r1` 和 `r2` 是 `p` 的任何两个装箱转换的结果, 则始终有 `r1 == r2`。

### 讨论

理想情况下, 装箱一个给定的基本值 `p`, 将始终会产生一个相同的引用。实际上, 使用现有的实现技术, 这可能不是切实可行的。上述规则是实际的折衷。上面的 `final` 子句总是需要把某些常用值装箱进不可区分的对象中。实现可能懒洋洋地或者急切地缓存这些对象。

对于其他的值, 从程序员的角度讲, 这种表述禁止任何关于装箱值的同一性假设。这将允许 (但不需要) 共享其中一些或所有这些引用。

在最常见的情况下, 将确保采取期待的行为, 而不会强加不适当的性能惩罚, 特别是在小型设备上则更是如此。例如, 对内存限制较少的实现可能缓存所有的字符或短整型的值, 以及 `-32K~+32K` 之间的整型和长整型的值。

如果某一包装器类 (`Boolean`、`Byte`、`Character`、`Short`、`Integer`、`Long`、`Float` 或 `Double`) 的新实例需要被分配并且可用内存不足, 那么装箱转换可能导致一个 `OutOfMemoryError`。

## 5.1.8 拆箱转换

拆箱转换把引用类型的值转换成相应的基本类型的值。确切地讲, 下列 8 种转换称为拆箱转换:

- 从类型 Boolean 到类型 boolean
- 从类型 Byte 到类型 byte
- 从类型 Character 到类型 char
- 从类型 Short 到类型 short
- 从类型 Integer 到类型 int
- 从类型 Long 到类型 long
- 从类型 Float 到类型 float
- 从类型 Double 到类型 double

在运行时, 拆箱转换的工作方式如下:

- 如果  $r$  是 Boolean 类型的引用, 那么拆箱转换将把  $r$  转换成  $r.booleanValue()$
- 如果  $r$  是 Byte 类型的引用, 那么拆箱转换将把  $r$  转换成  $r.byteValue()$
- 如果  $r$  是 Character 类型的引用, 那么拆箱转换将把  $r$  转换成  $r.charValue()$
- 如果  $r$  是 Short 类型的引用, 那么拆箱转换将把  $r$  转换成  $r.shortValue()$
- 如果  $r$  是 Integer 类型的引用, 那么拆箱转换将把  $r$  转换成  $r.intValue()$
- 如果  $r$  是 Long 类型的引用, 那么拆箱转换将把  $r$  转换成  $r.longValue()$
- 如果  $r$  是 Float 类型的引用, 那么拆箱转换将把  $r$  转换成  $r.floatValue()$
- 如果  $r$  是 Double 类型的引用, 那么拆箱转换将把  $r$  转换成  $r.doubleValue()$
- 如果  $r$  是 null, 那么拆箱转换将抛出一个 NullPointerException

如果某个类型是数值类型, 或者是可以通过拆箱转换来转换成数值类型的引用类型, 那么就称该类型可转换为数值类型。如果某个类型是整型, 或者是可以通过拆箱转换来转换成整型的引用类型, 则称该类型可转换为整型。

### 5.1.9 未经检查的转换

设  $G$  指定了一个泛型类型声明, 它带有  $n$  个形式类型参数。从原生类型 (4.8 节)  $G$  到  $G<T_1 \dots T_n>$  形式的任何参数化类型都有一个未经检查的转换。使用未经检查的转换将会生成一个强制性编译时警告 [只能使用 SuppressWarnings 注释 (9.6.1.5 节) 来禁止它], 除非参数化类型  $G$  是其中所有类型参数均为无界通配符 (4.5.1 节) 的参数化类型。



未经检查的转换用于支持在引入泛型类型之前编写的遗留代码与经历了使用泛型技术这一转换 [我们把这个过程称为泛型化 (generification)] 的库之间平滑的互操作性。

在这样的环境 (最明显的是 java.util 中的集合框架的客户) 下, 遗留代码使用原生类型 (例如, 用 Collection 代替 Collection<String>)。原生类型的表达式作为参数传递给库方法, 这些方法把那些相同类型的参数化版本用作它们相对应的形参的类型。

在使用泛型的类型系统之下, 这样的调用并不能证明是静态安全的。拒绝这样的调用将使现有代码中的一大部分无效, 并会阻止它们使用库的更新版本。这反过来又会使库供应商对于利用泛型技术感到气馁。

为了阻止这种情况的发生，可以把原生类型转换成该原生类型引用的泛型类型声明的任意调用。虽然这种转换是不合理的，但是出于对实用性让步的目的而容忍了它。在这种情况下，将会发出一个警告（称为未经检查的警告）。

### 5.1.10 捕获转换

设  $G$  指定了一个泛型类型声明，它带有  $n$  个形式类型参数  $A_1 \dots A_n$ ，它们具有的界限  $U_1 \dots U_n$ 。从  $G\langle T_1 \dots T_n \rangle$  到  $G\langle S_1 \dots S_n \rangle$  存在一个捕获转换，其中  $1 \leq i \leq n$ ：

- 如果  $T_i$  是  $?$  形式的通配符类型参数（4.5.1 节），那么  $S_i$  就是一个新类型变量，其上限为  $U_i[A_1 := S_1, \dots, A_n := S_n]$ ，下限为空类型。
- 如果  $T_i$  是  $? \text{ extends } B_i$  形式的通配符类型参数，那么  $S_i$  就是一个新类型变量，其上限为  $\text{glb}(B_i, U_i[A_1 := S_1, \dots, A_n := S_n])$ ，下限为空类型，其中  $\text{glb}(V_1, \dots, V_m)$  是  $V_1 \& \dots \& V_m$ 。如果对于任何两个类（不是接口） $V_i$  和  $V_j$ ， $V_i$  不是  $V_j$  的子类，或者反之亦然，那么就会引发编译时错误。
- 如果  $T_i$  是  $? \text{ super } B_i$  形式的通配符类型参数，那么  $S_i$  就是一个新类型变量，其上限为  $U_i[A_1 := S_1, \dots, A_n := S_n]$ ，下限为  $B_i$ 。
- 否则， $S_i = T_i$ 。

在除参数化类型（4.5 节）以外的任何类型上的捕获转换都充当同一性转换（5.1.1 节）。捕获转换从不需要在运行时执行特殊的动作，因而从来不会在运行时抛出异常。

不能递归地应用捕获转换。

#### 讨论

捕获转换旨在使通配符更有用。为了理解这个动机，我们首先来查看方法 `java.util.Collections.reverse()`：

```
public static void reverse(List<?> list);
```

该方法用于反转提供作为参数的列表。它可以用于任何列表类型，因此，把通配符类型 `List<?>` 用作形参的类型是完全合适的。

现在考虑如何实现 `reverse()`。

```
public static void reverse(List<?> list) { rev(list); }
private static <T> void rev(List<T> list) {
    List<T> tmp = new ArrayList<T>(list);
    for (int i = 0; i < list.size(); i++) {
        list.set(i, tmp.get(list.size() - i - 1));
    }
}
```

这个实现需要复制列表，从副本中提取元素，并把它们插入到原始列表中。为了以一种类型安全的方式执行这项任务，我们需要给传入列表的元素类型赋予一个名称  $T$ 。我们在私有服务方法 `rev()` 中执行这项任务。

这要求我们把类型 `List<?>` 的传入参数列表作为参数传递给 `rev()`。注意，一般来

讲, `List<?>` 是未知类型的列表。对于任何类型 `T`, 它不是 `List<T>` 的子类型。允许这样一种子类型关系是不合理的。给定以下方法:

```
public static <T> void fill(List<T> l, T obj)
```

下面的调用:

```
List<String> ls = new ArrayList<String>();
List<?> l = ls;
Collections.fill(l, new Object()); // not really legal -but assume
                                     // it was
String s = ls.get(0); // ClassCastException -ls contains Objects,
                       //not Strings.
```

将破坏类型系统。

因此, 如果不具有某个特殊的特许, 我们可以看到从 `reverse()` 到 `rev()` 的调用将被禁止。如果是这样的话, `reverse()` 的作者将被迫把其签名写为:

```
public static <T> void reverse(List<T> list)
```

这不是人们想要的, 因为它把实现信息向调用者公开了。更糟糕的是, API 的设计者可能推断: 使用通配符的签名就是 API 的调用者所需要的, 并且只在以后才意识到这阻止了类型安全的实现。

从 `reverse()` 到 `rev()` 的调用事实上是无害的, 但是不能基于 `List<?>` 和 `List<T>` 之间常规的子类型关系来证明它是合理的。该调用是无害的, 这是因为传入的参数无疑是某种类型 (虽然是未知的) 的列表。如果我們可以在类型变量 `x` 中捕获这种未知的类型, 就能够推断 `T` 是 `x`。这就是捕获转换的本质。当然, 本规范必须应付各种复杂的情况, 如重要的 (并且可能是递归定义的) 上限或下限, 存在多个参数的情况等。

## 讨论

数学知识渊博的读者希望将捕获转换与已建立的类型理论关联起来。不熟悉类型理论的读者可以跳过这个讨论——否则, 要学习合适的书籍, 如 Benjamin Pierce 所著的《Types and Programming Languages》, 然后重新阅读本部分内容。

这里简短地总结了捕获转换与已建立的类型理论概念的关系。

通配符类型是存在的类型的一种受限形式。捕获转换松散地对应于打开存在类型的值。表达式 `e` 的捕获转换可以被视作在包含封闭 `e` 的顶级表达式的作用域内打开 `e`。

存在情态上经典的 `open` 操作需要捕获的类型变量绝对禁止转义打开的表达式。对应于捕获转换的 `open` 总是作用在一个足够大的作用域上, 在该作用域的外部永远不能看到捕获的类型变量。

这种模式的优点是: 无需 `close` 操作, 就如 Atsushi Iga-rashi 和 Mirko Viroli 所著的论文《On Variance-Based Subtyping for Parametric Types》中所定义的那样, 该论文收集在关于面向对象编程的第 16 届欧洲大会 (ECOOP 2002 年) 的会议录中。

有关通配符的正式说明, 参见 Mads Torgersen、Erik Ernst 和 Christian Plesner 在面向对象编程基础 (FOOL 2005 年) 的第 12 次研讨会上所著的《Wild FJ》。

### 5.1.11 字符串转换

从任何其他类型（包括空类型）到 `String` 类型都存在字符串转换。参见第 5.4 节，以了解字符串转换环境的详细信息。

### 5.1.12 禁止的转换

不显式地允许的任何转换都是禁止的转换。

### 5.1.13 值集合转换

值集合转换是在不改变其类型的情况下把浮点值从一个值集映射到另一个值集合的过程。

在非精确浮点数（FP-strict）（15.4 节）的表达式内，值集转换为 Java 编程语言的实现提供了以下选择：

- 如果值是浮点扩展的指数值集的元素，那么实现可以选择把该值映射到浮点值集的最接近的元素。这种转换可能导致上溢（在这种情况下，该值将会被带有相同符号的无穷大取代）或下溢（在这种情况下，该值可能损失精度，因为它会被带有相同符号的非规范化的数或 0 取代）。
- 如果值是双精度扩展的指数值集的元素，那么实现可以选择把该值映射到双精度值集的最接近的元素。这种转换可能导致上溢（在这种情况下，该值将会被带有相同符号的无穷大取代）或下溢（在这种情况下，该值可能损失精度，因为它会被带有相同符号的非规范化的数或 0 取代）。

在精确浮点（FP-strict）的表达式（15.4 节）内，值集转换不会提供任何选择：每种实现都必须以相同的方式工作：

- 如果值是 `float` 类型的值，并且不是浮点值集的元素，那么实现必须把该值映射到浮点值集的最接近的元素。这种转换可能导致上溢或下溢。
- 如果值是 `double` 类型的值，并且不是双精度值集的元素，那么实现必须把该值映射到双精度值集的最接近的元素。这种转换可能导致上溢或下溢。

在精确浮点的表达式中，仅当调用一个其声明不是浮点精确的方法，并且实现选择把方法调用的结果表示为扩展的指数值集的元素时，从浮点扩展的指数值集或双精度扩展的指数值集映射值才是必要的。

无论是在浮点精确的代码中，还是在非浮点精确的代码中，对于其类型既不是 `float` 也不是 `double` 的任何值，值集转换始终会使其保持不变。

## 5.2 赋值转换

当把表达式的值赋予（15.26 节）一个变量时，就会发生赋值转换：必须把表达式的类型转换为变量的类型。赋值环境允许使用以下转换之一：

- 同一性转换（5.1.1 节）



- 扩展基本转换 (5.1.2 节)
- 扩展引用转换 (5.1.5 节)
- 装箱转换 (5.1.7 节), 可以选择后接扩展引用转换
- 拆箱转换 (5.1.8 节), 可以选择后接扩展基本转换

在应用了上面列出的转换之后, 如果得到的类型是原生类型 (4.8 节), 那么就可以应用未经检查的转换 (5.1.9 节)。如果转换链包含两个不在子类型关系中的参数化类型, 就会引发编译时错误。

#### 例 5.1

这样一个非法链的示例如下:

```
Integer, Comparable<Integer>, Comparable, Comparable<String>
```

该链中的前三个元素是通过扩展引用转换关联的, 而最后一项则通过未经检查的转换由其前驱派生而来。但是, 这不是一个有效的赋值转换, 因为该链包含两个参数化类型 `Comparable<Integer>` 和 `Comparable<String>`, 它们不是子类型。

此外, 如果表达式是类型 `byte`、`short`、`char` 或 `int` 的常量表达式 (15.28 节), 则:

- 如果变量的类型是 `byte`、`short` 或 `char`, 并且常量表达式的值在变量的类型中是可表示的, 就可以使用收缩基本转换。
- 如果满足以下条件, 则可以使用收缩基本转换, 后接装箱转换:
  - ◆ 变量的类型是 `Byte`, 并且常量表达式的值在类型 `byte` 中是可表示的。
  - ◆ 变量的类型是 `Short`, 并且常量表达式的值在类型 `short` 中是可表示的。
  - ◆ 变量的类型是 `Character`, 并且常量表达式的值在类型 `char` 中是可表示的。

如果不能通过赋值环境中允许的转换把表达式的类型转换成变量的类型, 那么就会发生编译时错误。

如果变量的类型是 `float` 或 `double`, 那么就会对值 `v` 应用值集转换, 其中 `v` 是类型转换的结果:

- 如果 `v` 是 `float` 类型, 并且是浮点扩展的指数值集的一个元素, 那么实现必须把 `v` 映射到浮点值集最接近的元素。这种转换可能导致上溢或下溢。
- 如果 `v` 是 `double` 类型, 并且是双精度扩展的指数值集的一个元素, 那么实现必须把 `v` 映射到双精度值集最接近的元素。这种转换可能导致上溢或下溢。

如果可以通过赋值转换, 把表达式的类型转换为变量的类型, 我们就称表达式 (或其值) 可以赋值给变量, 一种等价的说法是, 表达式的类型与变量的类型是赋值兼容的。

在应用了上述类型转换之后, 如果得到的值是一个变量, 并且该变量不是变量类型擦除的子类或子接口的实例, 那么就会抛出一个 `ClassCastException`。

仅当出现堆污染 (4.12.2.1 节) 时, 才会发生这种情形。



实际上，当字段的擦除类型或方法的擦除结果类型不同于它们的未擦除类型时，在访问参数化类型的对象的字段或方法时，实现只需要执行强制转换。

赋值转换可能引发的仅有的几个异常是：

- 装箱转换引发的 `OutOfMemoryError`。
- 上述特殊情形中的 `ClassCastException`。
- 空引用上的拆箱转换引发的 `NullPointerException`。

（但是注意，赋值可能导致涉及数组元素或字段访问的特殊情况中的异常——参见第 10.10 节和第 15.26.1 节。）

常量的编译时收缩意味着允许如下代码：

```
byte theAnswer = 42;
```

在不进行收缩的情况下，整数 42 具有类型 `int` 这一事实意味着将需要对 `byte` 进行强制转换：

```
byte theAnswer = (byte)42; // cast is permitted but not required
```

下面的测试程序包含基本值的赋值转换的示例：

```
class Test {
    public static void main(String[] args) {
        short s = 12;           // narrow 12 to short
        float f = s;            // widen short to float
        System.out.println("f=" + f);
        char c = '\u0123';
        long l = c;              // widen char to long
        System.out.println("l=0x" + Long.toString(l,16));
        f = 1.23f;
        double d = f;            // widen float to double
        System.out.println("d=" + d);
    }
}
```

它产生以下输出：

```
f=12.0
l=0x123
d=1.2300000190734863
```

但是，下列测试会产生编译时错误：

```
class Test {
    public static void main(String[] args) {
        short s = 123;
        char c = s; // error: would require cast
        s = c;      // error: would require cast
    }
}
```

这是由于并非所有的 `short` 值都是 `char` 值，同样，并非所有的 `char` 值都是 `short` 值。

可以把空类型的值（空引用是惟一种这样的值）赋予任何引用类型，从而得到那种类型的空引用。

下面的示例程序阐释了引用的赋值：

```
public class Point { int x, y; }
public class Point3D extends Point { int z; }
public interface Colorable {
    void setColor(int color);
}
public class ColoredPoint extends Point implements Colorable
{
    int color;
    public void setColor(int color) { this.color = color; }
}
class Test {
    public static void main(String[] args) {
        // Assignments to variables of class type:
        Point p = new Point();
        p = new Point3D(); // ok: because Point3D is a
                          // subclass of Point
        Point3D p3d = p; // error: will require a cast because a
                          // Point might not be a Point3D
                          // (even though it is, dynamically,
                          // in this example.)
        // Assignments to variables of type Object:
        Object o = p; // ok: any object to Object
        int[] a = new int[3];
        Object o2 = a; // ok: an array to Object
        // Assignments to variables of interface type:
        ColoredPoint cp = new ColoredPoint();
        Colorable c = cp; // ok: ColoredPoint implements
                          // Colorable
        // Assignments to variables of array type:
        byte[] b = new byte[4];
        a = b; // error: these are not arrays
               // of the same primitive type
        Point3D[] p3da = new Point3D[3];
        Point[] pa = p3da; // ok: since we can assign a
                           // Point3D to a Point
        p3da = pa; // error: (cast needed) since a Point
                  // can't be assigned to a Point3D
    }
}
```

下面的测试程序阐释了引用值上的赋值转换，但是它无法通过编译，如其注释所述。应该把本示例与前一个示例进行比较。

```
public class Point { int x, y; }
public interface Colorable { void setColor(int color); }
public class ColoredPoint extends Point implements Colorable
```

```

{
    int color;
    public void setColor(int color) { this.color = color; }
}

class Test {
    public static void main(String[] args) {
        Point p = new Point();
        ColoredPoint cp = new ColoredPoint();
        // Okay because ColoredPoint is a subclass of Point:
        p = cp;
        // Okay because ColoredPoint implements Colorable:
        Colorable c = cp;
        // The following cause compile-time errors because
        // we cannot be sure they will succeed, depending on
        // the run-time type of p; a run-time check will be
        // necessary for the needed narrowing conversion and
        // must be indicated by including a cast:
        cp = p; // p might be neither a ColoredPoint
                // nor a subclass of ColoredPoint
        c = p; // p might not implement Colorable
    }
}

```

下面是涉及数组对象赋值的另一个示例：

```

class Point { int x, y; }
class ColoredPoint extends Point { int color; }
class Test {
    public static void main(String[] args) {
        long[] veclong = new long[100];
        Object o = veclong; // okay
        Long l = veclong; // compile-time error
        short[] vecshort = veclong; // compile-time error
        Point[] pvec = new Point[100];
        ColoredPoint[] cpvec = new ColoredPoint[100];
        pvec = cpvec; // okay
        pvec[0] = new Point(); // okay at compile time,
                               // but would throw an
                               // exception at run time
        cpvec = pvec; // compile-time error
    }
}

```

在本示例中：

- 不能把 veclong 的值赋予 Long 变量，这是因为 Long 是类类型，而不是 Object。只能把数组赋予兼容的数组类型的变量，或者赋予 Object、Cloneable 或 java.io.Serializable 类型的变量。
- 不能把 veclong 的值赋予 vecshort，这是因为它们和基本类型的数组，而 short 和 long 不是相同的基本类型。

- 不能把 `cpvec` 的值赋予 `pvec`，这是因为可以作为 `ColoredPoint` 类型的表达式值的任何引用都可以作为 `Point` 类型的变量的值。这样，后来把新的 `Point` 赋予 `pvec` 的元素将抛出一个 `ArrayStoreException`（否则，如果校正程序以使之可以被编译），这是由于 `ColoredPoint` 数组不能把 `Point` 的实例作为元素的值。
- 不能把 `pvec` 的值赋予 `cpvec`，这是由于并非可以作为 `ColoredPoint` 类型的表达式值的所有引用都能够正确地作为 `Point` 类型的变量的值。如果在运行时 `pvec` 的值是指向 `Point[]` 的实例的引用，并且允许对 `cpvec` 赋值，那么指向 `cpvec` 的一个元素（比方说 `cpvec[0]`）的简单引用就可以返回一个 `Point`，并且这个 `Point` 不是一个 `ColoredPoint`。因此，为了允许这样一种赋值方式，则将允许违背类型系统。可以使用强制转换（5.5 节、15.16 节）以确保 `pvec` 引用 `ColoredPoint[]`：

```
cpvec = (ColoredPoint[])pvec;    // okay, but may throw an
                                // exception at run time
```

### 5.3 方法调用转换

方法调用转换适用于方法或构造函数调用（8.8.7.1 节、15.9 节、15.12 节）中的每个参数值：必须把参数表达式的类型转换成对应参数的类型。方法调用环境允许使用以下转换之一：

- 同一性转换（5.1.1 节）
- 扩展基本转换（5.1.2 节）
- 扩展引用转换（5.1.5 节）
- 装箱转换（5.1.7 节），可以选择后接扩展引用转换
- 拆箱转换（5.1.8 节），可以选择后接扩展基本转换。

在应用了上面列出的转换之后，如果得到的类型是原生类型（4.8 节），那么就可以应用一个未经检查的转换（5.1.9 节）。如果转换链包含两个不在子类型关系中的参数化类型，则会发生编译时错误。

如果参数表达式的类型是 `float` 或 `double`，那么在类型转换之后将应用值集转换（5.1.13 节）：

- 如果 `float` 类型的参数值是浮点扩展的指数值集的元素，那么实现必须把该值映射到浮点值集的最接近的元素。这种转换可能导致上溢或下溢。
- 如果 `double` 类型的参数值是浮点扩展的指数值集的元素，那么实现必须把该值映射到双精度值集的最接近的元素。这种转换可能导致上溢或下溢。

在应用了上述类型转换之后，如果得到的值是一个对象，并且该对象不是对应形参类型擦除的子类或子接口的实例，那么就会抛出一个 `ClassCastException`。




---

仅当出现堆污染（4.12.2.1 节）时，才会发生这种情形。

---

确切地讲，方法调用转换不包括作为赋值转换（5.2 节）一部分的整型常量的隐式收缩。Java 编程语言的设计者感到，包括这些隐式收缩转换将给匹配求解过程的重载方法增加额外的复杂性（15.12.2 节）。

因此，下面的示例：

```
class Test {  
    static int m(byte a, int b) { return a+b; }  
    static int m(short a, short b) { return a-b; }  
    public static void main(String[] args) {  
        System.out.println(m(12, 2)); // compile-time error  
    }  
}
```

将引发一个编译时错误，这是因为整数 12 和 2 具有 int 类型，因此方法 m 不符合第 15.12.2 节中的规则。包括整型常量隐式收缩的语言将需要额外的规则来解决像本示例这样的情况。

## 5.4 字符串转换

当参数之一是 String 时，才会对二元+运算符的操作数应用二元+运算符。在单独一个这样的特例中，+的另一个参数将被转换成 String，作为两个字符串串接的新 String 是+的结果。字符串串接+运算符的描述中详细说明了字符串转换（15.18.1 节）。

## 5.5 强制转换

歌声远离悲伤，命运远离关爱。  
——米格尔·德·塞万提斯（1547~1616），《堂吉珂德》

强制转换应用于强制转换运算符（15.16 节）的操作数：必须将操作数表达式的类型转换成强制转换运算符显式指定的类型。强制转换环境允许使用：

- 同一性转换（5.1.1 节）
- 扩展基本转换（5.1.2 节）
- 收缩基本转换（5.1.3 节）
- 扩展引用转换（5.1.5 节），可以选择后接未经检查的转换（5.1.9 节）
- 收缩引用转换（5.1.6 节），可以选择后接未经检查的转换
- 装箱转换（5.1.7 节）
- 拆箱转换（5.1.8 节）

因此，强制转换转换比赋值转换或方法调用转换更具包容性：强制转换转换可以执行除字符串转换或捕获转换（5.1.10 节）之外的任何允许的转换。

在类型转换之后应用值集转换（5.1.13 节）。

可以证明有些强制转换在编译时不正确：这些强制转换会导致编译时错误。

如果类型相同，则可以通过同一性转换，或者通过扩展基本转换或收缩基本转换把基

元类型的值强制转换成另一个基本类型。

通过装箱转换 (5.1.7 节), 可以把基本类型的值强制转换成引用类型。通过拆箱转换 (5.1.8 节), 可以把引用类型的值强制转换成基本类型。

余下的情况涉及把编译时引用类型  $S$  (源) 转换成编译时引用类型  $T$  (目标)。当且仅当  $S <: T$  (4.10 节) 时, 静态地可知从类型  $S$  到类型  $T$  的强制转换是正确的。从类型  $S$  到参数化类型 (4.5 节)  $T$  的强制转换是未经检查的, 除非下列条件中至少有一个成立:

- $S <: T$ 。
- $T$  的所有类型参数 (4.5.1 节) 都是无界通配符。
- $T <: S$  和  $S$  不具有子类型  $X \neq T$ , 这样,  $X$  和  $T$  的擦除 (4.6 节) 是相同的。

类型变量 (4.4 节) 的强制转换总是未经检查的。

如果静态地可知从  $|S|$  到  $|T|$  的强制转换是正确的, 那么从  $S$  到  $T$  的未经检查的强制转换是完全未经检查的。否则, 它就是部分未经检查的。未经检查的强制转换将引发未经检查的警告, 除非使用 `SuppressWarnings` 注释 (9.6.1.5 节) 禁止它。

如果一个强制转换不是静态地可知它是正确的, 并且它不是未经检查的, 那么它就是一个受查的强制转换。

从编译时引用类型  $S$  的值到编译时引用类型  $T$  的强制转换的编译时合法性的详细规则如下:

- 如果  $S$  是类类型:
  - ◆ 若  $T$  是类类型, 则有  $|S| <: |T|$  或  $|T| <: |S|$ ; 否则就会发生编译时错误。此外, 若存在  $T$  的超类型  $X$  和  $S$  的超类型  $Y$ , 并且可证明  $X$  和  $Y$  是截然不同的参数化类型 (4.5 节), 且  $X$  和  $Y$  的擦除是相同的, 那么就会发生编译时错误。
  - ◆ 若  $T$  是接口类型:
    - ◇ 如果  $S$  不是 `final` 类 (8.1.1 节), 则如果存在  $T$  的超类型  $X$  和  $S$  的超类型  $Y$ , 并且可证明  $X$  和  $Y$  是截然不同的参数化类型, 且  $X$  和  $Y$  的擦除是相同的, 那么就会发生编译时错误。否则, 强制转换在编译时总是合法的 (这是由于即使  $S$  不实现  $T$ ,  $S$  的子类也有可能实现它)。
    - ◇ 如果  $S$  是 `final` 类 (8.1.1 节), 那么  $S$  必须实现  $T$ , 否则就会发生编译时错误。
  - ◆ 若  $T$  是类型变量, 那么就会使用  $T$  的上限代替  $T$ , 递归地应用本算法。
  - ◆ 若  $T$  是数组类型, 那么  $S$  必须是类 `Object`, 否则就会发生编译时错误。
- 如果  $S$  是接口类型:
  - ◆ 若  $T$  是数组类型, 那么  $T$  必须实现  $S$ , 否则就会发生编译时错误。
  - ◆ 若  $T$  不是 `final` 类型 (8.1.1 节), 那么如果存在  $T$  的超类型  $X$  和  $S$  的超类型  $Y$ , 并且可证明  $X$  和  $Y$  是截然不同的参数化类型, 且  $X$  和  $Y$  的擦除是相同的, 那么就会发生编译时错误。否则, 强制转换在编译时总是合法的 (这是由于即使  $T$  不实现  $S$ ,  $T$  的子类也有可能实现它)。
  - ◆ 若  $T$  是 `final` 类型, 则:
    - ◇ 如果  $S$  不是参数化类型或原生类型, 那么  $T$  必须实现  $S$ , 并且静态地可知强制



转换是正确的，否则就会发生编译时错误。

- ✧ 否则， $S$  是作为某个泛型类型声明  $G$  的调用的参数化类型，或者是对应于泛型类型声明  $G$  的原生类型。那么必定存在  $T$  的超类型  $X$ ，并且  $X$  是  $G$  的调用，否则就会发生编译时错误。此外，如果可证明  $S$  和  $X$  是截然不同的参数化类型，那么就会发生编译时错误。

- 如果  $S$  是类型变量，那么就会使用  $S$  的上限代替  $S$ ，递归地应用本算法。
- 如果  $S$  是数组类型  $SC[]$ ，即类型  $SC$  的元素的数组：
  - ◆ 若  $T$  是类类型，那么若  $T$  不是 `Object`，则会发生编译时错误（由于 `Object` 是可以给其赋予数组的惟一的类类型）。
  - ◆ 若  $T$  是接口类型，除非  $T$  是类型 `java.io.Serializable` 或类型 `Cloneable`，即被数组实现的惟一两个接口，否则就会发生编译时错误。
  - ◆ 若  $T$  是类型变量，则：
    - ✧ 如果  $T$  的上限是 `Object` 或类型 `java.io.Serializable` 或类型 `Cloneable`，亦或是可以通过递归地应用这些规则合法地强制转换为  $S$  的类型变量，那么强制转换就是合法的（尽管是未经检查的）。
    - ✧ 如果  $T$  的上限是数组类型  $TC[]$ ，那么除非可以通过递归应用这些强制转换的编译时规则把类型  $SC[]$  强制转换成  $TC[]$ ，否则就会发生编译时错误。
    - ✧ 否则，就会发生编译时错误。
  - ◆ 若  $T$  是数组类型  $TC[]$ ，即类型  $TC$  的元素的数组，那么除非以下条件之一成立，否则就会发生编译时错误：
    - ✧  $TC$  和  $SC$  是相同的基本类型。
    - ✧  $TC$  和  $SC$  都是引用类型，并且可以通过递归应用这些强制转换的编译时规则把类型  $SC$  强制转换成  $TC$ 。

分别参见第 8 章、第 9 章和第 10 章，以查看有关类、接口和数组的规范。

如果强制转换到引用类型不是编译时错误，则有以下几种情形：

- 静态地可知强制转换是正确的。对于这种强制转换，不会执行任何运行时动作。
- 强制转换是完全未经检查的强制转换。对于这种强制转换，不会执行任何运行时动作。
- 强制转换是部分未经检查的强制转换。这种强制转换需要运行时有效性检查。该检查的执行方式是：强制转换好像是  $|S|$  和  $|T|$  之间经过检查的强制转换，如下所述。
- 强制转换是经过检查的强制转换。这种强制转换需要运行时有效性检查。如果运行时的值是 `null`，那么就允许这种强制转换。否则，设  $R$  是被运行时引用值引用的对象的类，并且设  $T$  是强制转换运算符中指定类型的擦除。在运行时，强制转换必须检查类  $R$  与类型  $T$  是赋值兼容的（注意，当最初为任何给定的强制转换应用这些规则时， $R$  不能是接口，但是如果递归地应用这些规则，则  $R$  可以是一个接口，这是由于运行时引用值可能引用其元素类型是接口类型的数组）。用于执行检查的算法如下所示：
  - ◆ 若  $R$  是一个普通类（而不是一个数组类）：

- ◇ 如果  $T$  是类类型, 那么  $R$  必须是与  $T$  或  $T$  的子类相同的类 (4.3.4 节), 否则就会抛出一个运行时异常。
- ◇ 如果  $T$  是接口类型, 那么  $R$  必须实现 (8.1.5 节) 接口  $T$ , 否则就会抛出一个运行时异常。
- ◇ 如果  $T$  是数组类型, 那么就会抛出一个运行时异常。
- ◆ 若  $R$  是一个接口:
  - ◇ 如果  $T$  是类类型, 那么  $T$  必须是 `Object` (4.3.2 节), 否则就会抛出一个运行时异常。
  - ◇ 如果  $T$  是接口类型, 那么  $R$  必须与  $T$  或  $T$  的子接口相同的接口, 否则就会抛出一个运行时异常。
  - ◇ 如果  $T$  是数组类型, 那么就会抛出一个运行时异常。
- ◆ 若  $R$  是表示一个数组类型  $RC[]$  (即类型  $RC$  的元素的数组) 的类:
  - ◇ 如果  $T$  是类类型, 那么  $T$  必须是 `Object` (4.3.2 节), 否则就会抛出一个运行时异常。
  - ◇ 如果  $T$  是接口类型, 那么除非  $T$  是类型 `java.io.Serializable` 或类型 `Cloneable`, 即由数组实现的惟一两个接口, 否则就会抛出一个运行时异常 (例如, 如果如果一个指向数组的引用存储在 `Object` 类型的变量中, 那么这种情况可能错过传递编译时检查)。
  - ◇ 如果  $T$  是数组类型  $TC[]$ , 即类型  $TC$  的元素的数组, 那么除非以下条件之一成立, 否则就会抛出一个运行时异常:
    - $TC$  和  $RC$  是相同的基本类型。
    - $TC$  和  $RC$  都是引用类型, 并且可以通过递归应用强制转换的这些运行时规则把类型  $RC$  强制转换成  $TC$ 。

如果抛出一个运行时异常, 则它是一个 `ClassCastException`。

下面给出了引用类型的强制转换的一些示例, 它们类似于第 5.2 节中的示例:

```
public class Point { int x, y; }
public interface Colorable { void setColor(int color); }
public class ColoredPoint extends Point implements Colorable
{
    int color;
    public void setColor(int color) { this.color = color; }
}
final class EndPoint extends Point { }
class Test {
    public static void main(String[] args) {
        Point p = new Point();
        ColoredPoint cp = new ColoredPoint();
        Colorable c;
        // The following may cause errors at run time because
        // we cannot be sure they will succeed; this possibility
```

```
// is suggested by the casts:
cp = (ColoredPoint)p;    // p might not reference an
                        // object which is a ColoredPoint
                        // or a subclass of ColoredPoint
c = (Colorable)p;        // p might not be Colorable
// The following are incorrect at compile time because
// they can never succeed as explained in the text:
Long l = (Long)p;        // compile-time error #1
EndPoint e = new EndPoint();
c = (Colorable)e;        // compile-time error #2
}
}
```

由于类类型 Long 和 Point 不相关（也就是说，它们不相同，并且它们的子类也不相同），因此它们之间的强制转换总是会失败，所以这里会发生第一个编译时错误。

由于 EndPoint 类型的变量永远不可能引用一个实现 Colorable 接口的值，所以会发生第二个编译时错误。这是由于 EndPoint 是 final 类型，并且 final 类型的变量总是会存储一个与其编译时类型相同的运行时类型的值。因此，变量 e 的运行时类型必须正好是类型 EndPoint，并且类型 EndPoint 不实现 Colorable。

下面是一个涉及数组（第 10 章）的示例：

```
class Point {
    int x, y;
    Point(int x, int y) { this.x = x; this.y = y; }
    public String toString() { return "("+x+","+y+")"; }
}
public interface Colorable { void setColor(int color); }
public class ColoredPoint extends Point implements Colorable
{
    int color;
    ColoredPoint(int x, int y, int color) {
        super(x, y); setColor(color);
    }
    public void setColor(int color) { this.color = color; }
    public String toString() {
        return super.toString() + "@" + color;
    }
}
class Test {
    public static void main(String[] args) {
        Point[] pa = new ColoredPoint[4];
        pa[0] = new ColoredPoint(2, 2, 12);
        pa[1] = new ColoredPoint(4, 5, 24);
        ColoredPoint[] cpa = (ColoredPoint[])pa;
        System.out.print("cpa: {");
        for (int i = 0; i < cpa.length; i++)
```

```
        System.out.print((i == 0 ? " " : ", ") + cpa[i]);
        System.out.println(" ");
    }
}
```

本示例将无错地通过编译，并产生以下输出：

```
cpa: { (2,2)@12, (4,5)@24, null, null }
```

下面的示例使用强制转换进行编译，但是它会在运行时抛出异常，因为其类型不兼容：

```
public class Point { int x, y; }
public interface Colorable { void setColor(int color); }
public class ColoredPoint extends Point implements Colorable
{
    int color;
    public void setColor(int color) { this.color = color; }
}
class Test {
    public static void main(String[] args) {
        Point[] pa = new Point[100];
        // The following line will throw a ClassCastException:
        ColoredPoint[] cpa = (ColoredPoint[])pa;
        System.out.println(cpa[0]);
        int[] shortvec = new int[2];
        Object o = shortvec;
        // The following line will throw a ClassCastException:
        Colorable c = (Colorable)o;
        c.setColor(0);
    }
}
```

## 5.6 数值提升

数值提升被应用于算术运算符的操作数。数值提升环境允许使用同一性转换（5.1.1节）、扩展基本转换（5.1.2节）或拆箱转换（5.1.8节）。

数值提升用于把数值运算符的操作数转换成通用类型，以使得运算可以执行。数值提升有两种：一元数值提升（5.6.1节）和二元数值提升（5.6.2节）。

### 5.6.1 一元数值提升

有些运算符对单一操作数应用一元数值提升，这必定会产生一个数值类型的值：

- 如果操作数是编译时类型 `Byte`、`Short`、`Character` 或 `Integer`，则它服从拆箱转换。这样，就会通过扩展转换（5.1.2节）或同一性转换，把结果提升到 `int` 类型的值。
- 否则，如果操作数是编译时类型 `Long`、`Float` 或 `Double`，则它服从拆箱转换。

- 否则，如果操作数是编译时类型 `byte`、`short` 或 `char`，则一元数值提升会通过扩展转换（5.1.2 节），将其提升到一个 `int` 类型的值。
- 否则，一元数值操作数会保持不变，并且不会被转换。

无论对于哪一种情况，随后都会应用值集转换（5.1.13 节）。

在下列情况下，将会在表达式上执行一元数值提升：

- 数组创建表达式（15.10 节）中的每个维数表达式
- 数组访问表达式（15.13 节）中的索引表达式
- 一元加法运算符 `+`（15.15.3 节）的操作数
- 一元减法运算符 `-`（15.15.4 节）的操作数
- 逐位求补运算符 `~`（15.15.5 节）的操作数
- 移位运算符 `>>`、`>>>` 或 `<<`（15.19 节）的每个单独的操作数；因此 `long` 移位（右操作数）不会把被移位的值（左操作数）提升到 `long`

下面是一个测试程序，它包括一元数值提升的示例：

```
class Test {
    public static void main(String[] args) {
        byte b = 2;
        int a[] = new int[b];    // dimension expression promotion
        char c = '\u0001';
        a[c] = 1;                // index expression promotion
        a[0] = -c;               // unary -promotion
        System.out.println("a: " + a[0] + ", " + a[1]);
        b = -1;
        int i = ~b;              // bitwise complement promotion
        System.out.println("~0x" + Integer.toHexString(b)
            + " == 0x" + Integer.toHexString(i));
        i = b << 4L;            // shift promotion (left operand)
        System.out.println("0x" + Integer.toHexString(b)
            + " << 4L == 0x" + Integer.toHexString(i));
    }
}
```

本测试程序产生如下输出：

```
a: -1, 1
~0xffffffff == 0x0
0xffffffff << 4L == 0xffffffff0
```

## 5.6.2 二元数值提升

当运算符对一对操作数应用二元数值提升时，其中每个操作数都必须指定一个可以转换成数值类型的值，使用扩展转换（5.1.2 节）按顺序应用下列规则，以根据需要转换操作数：

- 如果任何一个操作数是引用类型，则执行拆箱转换（5.1.8 节）。然后：
- 如果有一个操作数是 `double` 类型，则把另一个操作数转换成 `double`。

- 否则，如果有一个操作数是 float 类型，则把另一个操作数转换成 float。
- 否则，如果有一个操作数是 long 类型，则把另一个操作数转换成 long。
- 否则把两个操作数都转换成 int 类型。

在类型转换（如果有的话）之后，将为每个操作数应用值集转换（5.1.13 节）。会对某些运算符的操作数执行二元数值提升：

- 乘法运算符\*、/和%（15.17 节）
- 用于数值类型的加法和减法运算符+和-（15.18.2 节）
- 数值比较运算符：<、<=、>和>=（15.20.1 节）
- 数值相等性运算符：==和!=（15.21.1 节）
- 整数逐位运算符&、^和|（15.22.1 节）
- 某些情况下的条件运算符？：（15.25 节）

在上面的第 5.1 节中给出了二元数值提升的一个示例。下面给出另一个示例：

```
class Test {
    public static void main(String[] args) {
        int i = 0;
        float f = 1.0f;
        double d = 2.0;
        // First int*float is promoted to float*float, then
        // float==double is promoted to double==double:
        if (i * f == d)
            System.out.println("oops");
        // A char&byte is promoted to int&int:
        byte b = 0x1f;
        char c = 'G';
        int control = c & b;
        System.out.println(Integer.toHexString(control));
        // Here int:float is promoted to float:float:
        f = (b==0) ? i : 4.0f;
        System.out.println(1.0/f);
    }
}
```

输出结果如下：

```
7
0.25
```

本示例通过屏蔽字符的除低 5 位以外的其他所有位，把 ASCII 字符 G 转换成 ASCII control-G (BEL)，7 是这个控制字符的数值。

啊，太阳哟！啊，墓边的青草哟！啊，永恒的转变和前进哟！

——沃尔特·惠特曼，《草叶集》



## 名 称

道可道，非常道；名可名，非常名。无名天地之始，有名万物之母。  
——老子《道德经》，公元前6世纪

名称用于指代程序中声明的实体。声明的实体（6.1节）有：包、类类型（常规或枚举）、接口类型（常规或注释类型）、引用类型的成员（类、接口、字段或方法）、（类、接口、方法或构造函数的）类型参数（4.4节）、（传递给方法、构造函数或异常处理程序的）参数或局部变量。

程序中的名称可以是简单名称（包括单一标识符）或限定名称，包括由“.”标记（6.2节）隔开的一系列标识符。

引入名称的每个声明都有一个作用域（6.3节），它是程序文本的一部分，在该程序文本内，可以通过简单名称来指代声明的实体。

包和引用类型（即类类型、接口类型和数组类型）具有成员（6.4节）。可以使用限定名称  $N.x$  来指代成员，其中  $N$  是简单或限定名称， $x$  是标识符。如果  $N$  指定了一个包，则  $x$  就是那个包的成员，它可以是类或接口类型，也可以是子包。如果  $N$  指定了一个引用类型或者引用类型的变量，那么  $x$  就指定了那个类型的成员，它可以是类、接口、字段或方法。

在确定名称的含义（6.5节）时，可以使用名称出现的环境来消除同名的包、类型、变量和方法之间的歧义。

可以在类、接口、方法或字段声明中指定访问控制（6.6节），来控制允许访问成员的时间。访问是一个不同于作用域的概念：访问指定了程序文本的某个部分，在这个部分内，可以通过限定名称、字段访问表达式（15.11节）或方法调用表达式（15.12节）来指代声明的实体，其中在方法调用表达式中，方法不是通过简单名称指定的。默认访问是在包含其声明的包内的任何位置访问成员；其他可能的访问方式有 `public`、`protected` 和 `private`。

本章还将讨论完全限定名和规范名称（6.7节）以及命名约定。

字段、参数或局部变量的名称可以用作表达式（15.14.2节）。方法的名称可以出现在只作为方法调用表达式（15.12节）一部分的表达式中。类或接口类型的名称可以出现在只作为类字面常数（class literal）（15.8.2节）、限定性 `this` 表达式（15.8.4节）、类实例创建表

达式 (15.9 节)、数组创建表达式 (15.10 节)、强制转换表达式 (15.16 节)、instanceof 表达式 (15.20.2 节)、枚举常量 (8.9 节), 或者作为字段或方法的限定名称一部分的表达式中。包的名称可以出现在只作为类或接口类型的限定名称一部分的表达式中。

## 6.1 声明

声明在程序中引入了一个实体, 并且包括可用于在名称中指代该实体的标识符 (3.8 节)。声明的实体是以下实体之一:

- 包, 在 package 声明 (7.4 节) 中声明
- 导入类型, 在单一类型导入声明 (7.5.1 节) 中或按需类型导入声明 (7.5.2 节) 中声明。
- 类, 在类类型声明 (8.1 节) 中声明
- 接口, 在接口类型声明 (9.1 节) 中声明
- 类型变量 (4.4 节), 声明为泛型类 (8.1.2 节)、接口 (9.1.2 节)、方法 (8.4.4 节) 或构造函数的形式类型参数 (8.8.1 节)
- 引用类型 (8.2 节、9.2 节、10.7 节) 的成员, 可以是以下之一:
  - ◆ 成员类 (8.5 节、9.5 节)
  - ◆ 成员接口 (8.5 节、9.5 节)
  - ◆ 枚举常量 (8.9 节)
  - ◆ 字段, 可以是以下之一:
    - ◇ 在类类型中声明的字段 (8.3 节)
    - ◇ 在接口类型中声明的常量字段 (9.3 节)
    - ◇ 字段 length, 它隐式地是每个数组类型的成员 (10.7 节)
  - ◆ 方法, 可以是以下之一:
    - ◇ 在类类型中声明的方法 (abstract 或其他类型) (8.4 节)
    - ◇ 在接口类型中声明的方法 (总是 abstract 类型) (9.4 节)
- 参数, 可以是以下之一:
  - ◆ 类的方法或构造函数的参数 (8.4.1 节、8.8.1 节)
  - ◆ 接口的 abstract 方法的参数 (9.4 节)
  - ◆ 在 try 语句的 catch 子句中声明的异常处理程序的参数 (14.20 节)
- 局部变量, 可以是以下之一:
  - ◆ 在块语句中声明的局部变量 (14.4 节)
  - ◆ 在 for 语句中声明的局部变量 (14.14 节)

构造函数 (8.8 节) 也是通过声明引入的, 但是使用类的名称 (而不是引入一个新名称) 来声明它们。

## 6.2 名称和标识符

名称用于指代程序中声明的实体。

有两种形式的名称：简单名称和限定名称。简单名称是单一标识符。限定名称由名称、“.”标记和标识符组成。

在确定名称的含义（6.5 节）时，需要考虑名称出现的环境。第 6.5 节的规则区分了名称必须指示（指代）包（6.5.3 节）、类型（6.5.5 节）、表达式中的变量或值（6.5.6 节），或者方法（6.5.7 节）的环境。

并非程序中的所有标识符都是名称的一部分。标识符也可用于下列情形：

- 在声明（6.1 节）中，其中出现的标识符可能指定一个名称，该名称用于知道声明的实体。
- 在字段访问表达式（15.11 节）中，其中出现在“.”标记后面的标识符用于指定对象的成员，该对象是出现在“.”标记前面的表达式或 `super` 关键字的值。
- 在某些方法调用表达式（15.12 节）中，其中出现在“.”标记后面和“(”标记前面的标识符用于指定要为某个对象调用的方法，该对象是出现在“.”标记前面的表达式或 `super` 关键字的值。
- 在限定性类实例创建表达式（15.9 节）中，其中紧接着最左边的 `new` 标记的右边出现的标识符用于指定必须作为主表达式的编译时类型成员的类型，该表达式位于最左边的 `new` 标记之前的“.”标记之前。
- 作为标签语句（14.7 节）以及 `break`（14.15 节）和 `continue`（14.16 节）语句中的标签，用于指代语句标签。

在下面的示例中：

```
class Test {
    public static void main(String[] args) {
        Class c = System.out.getClass();
        System.out.println(c.toString().length() +
                           args[0].length() + args.length);
    }
}
```

标识符 `Test`、`main` 以及第一次出现的 `args` 和 `c` 都不是名称：它们在声明中用于指定声明的实体的名称。示例中出现了 `String`、`Class`、`System.out.getClass`、`System.out.println`、`c.toString`、`args` 和 `args.length` 这些名称。第一次出现的 `length` 不是一个名称，而是出现在方法调用表达式（15.12 节）中的一个标识符。第二次出现的 `length` 也不是一个名称，它同样是出现在方法调用表达式（15.12 节）中的一个标识符。

Java 编程语言将标签语句及其关联的 `break` 和 `continue` 语句中使用的标识符与声明中使用的那些标识符完全隔离开。因此，下列代码是有效的：

```
class TestString {
    char[] value;
    int offset, count;
    int indexOf(TestString str, int fromIndex) {
        char[] v1 = value, v2 = str.value;
        int max = offset + (count - str.count);
        int start = offset + ((fromIndex < 0) ? 0 : fromIndex);
    }
}
```

```

    for (int i = start; i <= max; i++)
    {
        int n = str.count, j = i, k = str.offset;
        while (n-- != 0) {
            if (v1[j++] != v2[k++])
                continue i;
        }
        return i - offset;
    }
    return -1;
}
}

```

这段代码取自于类 `String` 及其方法 `indexOf` 的一个版本，其中的标签最初被称为 `test`。把该标签更改为具有与局部变量 `i` 相同的名称不会模糊（obscure）（6.3.2 节）`i` 的声明作用域中的标签。标识符 `max` 也可用作语句标签；该标签不会模糊标签语句内的局部变量 `max`。

## 6.3 声明的作用域

声明的作用域是程序的作用区域，其中通过声明声明的实体可以使用简单名称来指代[假定它是可见的（6.3.1 节）]。当且仅当声明的作用域包括某个特殊点时，才称声明位于程序中那个点处的作用域中。

本章在描述一些构造的节中介绍了这些构造的作用域规则。为方便起见，这里重复介绍了这些规则：

可观察的（7.4.3 节）顶级包的声明的作用域是所有可观察的编译单元（7.3 节）。不可观察的包的声明永远不在作用域内。子包声明永远不在作用域内。

通过单一类型导入声明（7.5.1 节）或按需类型导入声明（7.5.2 节）导入的类型的的作用域，是导入声明出现的编译单元中的所有类和接口类型声明（7.6 节）。

通过单一静态导入声明（7.5.3 节）或按需静态导入声明（7.5.4 节）导入的成员的作用域，是导入声明出现的编译单元中的所有类和接口类型声明（7.6 节）。

顶级类型的作用域是在其中声明顶级类型的包中的所有类型声明。

在类类型 `C` 中声明或被 `C` 继承的成员 `m` 的声明的作用域是 `C` 的整个主体，包括任何嵌套的类型声明。

在任何接口类型 `I` 中声明或被 `I` 继承的成员 `m` 的声明的作用域是 `I` 的整个主体，包括任何嵌套的类型声明。

方法（8.4.1 节）或构造函数（8.8.1 节）的参数的作用域是方法或构造函数的整个主体。

接口的类型参数的作用域是接口的整个声明，包括类型参数部分本身。因此，类型参数可以作为其自己界限的某些部分出现，或者作为在相同部分中声明的其他类型参数的界限。

方法的类型参数的作用域是方法的整个声明，包括类型参数部分本身。因此，类型参

数可以作为其自己界限的某些部分出现，或者作为在相同部分中声明的其他类型参数的界限。

构造函数的类型参数的作用域是构造函数的整个声明，包括类型参数部分本身。因此，类型参数可以作为其自己界限的某些部分出现，或者作为在相同部分中声明的其他类型参数的界限。

块语句（14.4.2 节）中的局部变量声明的作用域是声明出现的块语句的其余部分，开始于它自己的初始化语句（14.4 节），并且包括位于局部变量声明语句右边的任何更多的声明符。

直接封闭在块语句（14.2 节）中的本地类的作用域是直接封闭的块语句的其余部分，包括它自己的类声明。直接封闭在 `switch` 块语句组（14.11 节）中的本地类的作用域是直接封闭的 `switch` 块语句组的其余部分，包括它自己的类声明。

在基本 `for` 语句（14.14 节）的 *ForInit* 部分中声明的局部变量的作用域包括下面所有这些方面：

- 它自己的初始化语句
- 出现在 `for` 语句的 *ForInit* 部分右边的任何更多声明符
- `for` 语句的表达式和 *ForUpdate* 部分
- 包含的语句

增强型 `for` 语句（14.14 节）的 *FormalParameter* 部分声明的局部变量的作用域是包含的语句。

`try` 语句（14.20 节）的 `catch` 子句中声明的异常处理程序的参数的作用域是与 `catch` 关联的整个块语句。

这些规则暗示在使用类型之前，不必出现类和接口类型的声明。

在下面的示例中：

```
package points;
class Point {
    int x, y;
    PointList list;
    Point next;
}
class PointList {
    Point first;
}
```

在类 `Point` 中使用 `PointList` 是正确的，这是因为类声明 `PointList` 的作用域包括类 `Point` 和类 `PointList`，以及包 `points` 的其他编译单元中任何其他类型声明。

### 6.3.1 屏蔽 (shadowing) 声明

有些声明可能会在其作用域的某些部分中被具有相同名称的另一个声明屏蔽，在这种情况下，不能使用简单名称指代声明的实体。

名为 *n* 的类型的声明 *d* 屏蔽了在 *d* 的整个作用域内出现 *d* 的那一点的作用域中任何其



他名为  $n$  的类型的声明。

名为  $n$  的字段、局部变量、方法参数、构造函数参数或异常处理程序参数的声明  $d$ ，屏蔽了在  $d$  的整个作用域内出现  $d$  的那一点的作用域中任何其他名为  $n$  的字段、局部变量、方法参数、构造函数参数或异常处理程序参数的声明。

名为  $n$  的方法的声明  $d$ ，屏蔽了在  $d$  的整个作用域内出现  $d$  的那一点的封闭作用域中任何其他名为  $n$  的方法的声明。

包声明永远不会屏蔽任何其他的声明。

对于包  $p$  的编译单元  $c$  中的单一类型导入声明  $d$ ，如果它导入名为  $n$  的类型，则将会屏蔽整个  $c$  中以下类型的声明：

- 在  $p$  的另一个编译单元中声明的名为  $n$  的任何顶级类型。
- 在  $c$  中通过按需类型导入声明导入的名为  $n$  的任何类型。
- 在  $c$  中通过按需静态导入声明导入的名为  $n$  的任何类型。

对于包  $p$  的编译单元  $c$  中的单一静态导入声明  $d$ ，如果它导入名为  $n$  的字段，则会在整个  $c$  中屏蔽  $c$  内通过按需静态导入声明导入的名为  $n$  的任何静态字段的声明。

对于包  $p$  的编译单元  $c$  中的单一静态导入声明  $d$ ，如果它导入具有签名  $s$  且名为  $n$  的方法，则会在整个  $c$  中屏蔽  $c$  内通过按需静态导入声明导入的具有签名  $s$  且名为  $n$  的任何静态方法的声明。

对于包  $p$  的编译单元  $c$  中的单一静态导入声明  $d$ ，如果它导入名为  $n$  的类型，则将会屏蔽整个  $c$  中以下类型的声明：

- 在  $c$  中通过按需静态导入声明导入的名为  $n$  的任何静态类型。
- 在  $p$  的另一个编译单元（7.3 节）中声明的名为  $n$  的任何顶级类型（7.6 节）。
- 在  $c$  中通过按需类型导入声明（7.5.2 节）导入的名为  $n$  的任何类型。

按需类型导入声明从来不会导致任何其他的声明被屏蔽。按需静态导入声明也从来不会导致任何其他的声明被屏蔽。

如果声明  $d$  的作用域包括程序中的某一点  $p$ ，则称  $d$  在  $p$  点可见，并且  $d$  不会被  $p$  点的任何其他声明屏蔽。当我们讨论的程序中的点对上下文清晰时，我们通常将简单地称声明是可见的。

注意，屏蔽不同于隐藏（hiding）（8.3 节、8.4.8.2 节、8.5 节、9.3 节、9.5 节）。从技术意义上讲，本规则中定义的隐藏仅适用于那些将被继承、但是由于子类中的声明而未被继承的成员。屏蔽也不同于模糊（obscuring）（6.3.2 节）。

下面是一个通过局部变量声明来屏蔽字段声明的示例：

```
class Test {
    static int x = 1;
    public static void main(String[] args) {
        int x = 0;
        System.out.print("x=" + x);
        System.out.println(", Test.x=" + Test.x);
    }
}
```



产生如下输出：

```
x=0, Test.x=1
```

本示例声明了：

- 类 Test
- 类 (static) 变量 x，它是类 Test 的成员
- 类方法 main，它是类 Test 的成员
- main 方法的参数 args
- main 方法的局部变量 x

由于类变量的作用域包括类的整个主体 (8.2 节)，因此类变量 x 通常将在方法 main 的整个主体中可用。但是在本示例中，在 main 方法体内，类变量 x 会被局部变量 x 的声明屏蔽。

局部变量将在其中声明它的块语句的其余部分作为其作用域 (14.4.2 节)；在这里，它是 main 方法体的其余部分，即它的初始值设定项 “0” 以及 print 和 println 的调用。

这意味着：

- Print 调用中的表达式 “x” 指代 (指示) 局部变量 x 的值。
- Println 的调用使用限定名称 (6.6 节) Test.x，它使用类类型名称 Test 访问类变量 x，这是由于 Test.x 的声明在这一点被屏蔽，并且不能被其简单名称指代。

下面的示例阐释了一个类型声明被另一个类型声明屏蔽：

```
import java.util.*;
class Vector {
    int val[] = { 1, 2 };
}
class Test {
    public static void main(String[] args) {
        Vector v = new Vector();
        System.out.println(v.val[0]);
    }
}
```

编译并输出：

```
1
```

相比可能按需导入的泛型 (8.1.2 节) 类 java.util.Vector，这里将优先使用声明的类 Vector。

### 6.3.2 模糊的声明

简单名称可能出现在它可能被潜在地解释为变量、类型或包的名称的环境中。在这些情形中，第 6.5 节的规则指定变量将优先于类型被选择，而类型将优先于包被选择。因此，有时也许不能通过其简单名称指代可见类型或包声明。我们称这种声明是模糊的 (obscured)。

模糊不同于屏蔽 (6.3.1 节) 和隐藏 (8.3 节、8.4.8.2 节、8.5 节、9.3 节、9.5 节)。第 6.8 节的命名约定有助于减少模糊性。

## 6.4 成员和继承

包和引用类型具有成员。

本节在这里概述了包和引用类型的成员，作为讨论限定名称和确定名称含义的背景。有关成员关系的完整描述，请参见第4.4节、4.5.2节、4.8节、4.9节、7.1节、8.2节、9.2节和10.7节。

### 6.4.1 类型变量、参数化类型、原生类型和交集类型的成员

第4.4节中指定了类型变量的成员；第4.5.2节中指定了参数化类型的成员；第4.8节中指定了原生类型的成员；第4.9节中则指定了交集类型的成员。

### 6.4.2 包的成员

第7.1节中指定了包（第7章）的成员。为方便起见，我们在这里重复介绍了该规范：

包的成员包括它的子包，以及包的所有编译单元（7.3节）中声明的所有顶级（7.6节）类类型（第8章）和顶级接口类型（第9章）。

一般来讲，包的子包是由主机系统（7.2节）确定的。但是，包 `java` 总是包括子包 `lang` 和 `io`，并且可能包括其他的子包。同一个包的两个不同的成员不能具有相同的简单名称（7.1节），但是不同包的成员可能具有相同的简单名称。

例如，有可能声明下面一个包：

```
package vector;  
public class Vector { Object[] vec; }
```

它具有一个名为 `Vector` 的 `public` 类作为成员，即使包 `java.util` 也声明了一个名为 `Vector` 的类也是如此。这两个类类型是不同的，它们具有不同的完全限定名称（6.7节）反映了这个事实。这个示例 `Vector` 的完全限定名称是 `vector.Vector`，而 `java.util.Vector` 是 Java 平台中通常包括的 `Vector` 类的完全限定名称。由于包 `vector` 包含一个名为 `Vector` 的类，因此它不能同时具有一个名为 `Vector` 的子包。

### 6.4.3 类类型的成员

类类型（8.2节）的成员有：类（8.5节、9.5节）、接口（8.5节、9.5节）、字段（8.3节、9.3节、10.7节）和方法（8.4节、9.4节）。成员可以是在类型中声明，或者是继承而来的，由于它们是超类或超接口的可访问成员，这些超类或超接口不会是私有的、隐藏的或重写的（8.4.8节）。

类类型的成员包括下面所有这些成员：

- 从其直接超类（8.1.4节）继承的成员，如果类有一个直接超类的话（类 `Object` 不具有直接超类）
- 从任何直接超接口（8.1.5节）继承的成员

- 在类体 (8.1.6 节) 中声明的成员

构造函数 (8.8 节) 和类型变量 (4.4 节) 都不是成员。对于类类型的字段和方法, 不限制它们具有相同的简单名称。同样, 对于类类型的成员类或成员接口, 也不限制它们具有与该类类型的字段或方法相同的简单名称。

如果类是在不同的接口中声明的, 并且继承自不同的接口, 那么该类可能有两个或多个具有相同简单名称的字段。尝试通过其简单名称指代任何字段, 将会导致编译时错误 (6.5.7 节、8.2 节)。

在下面的示例中:

```
interface Colors {
    int WHITE = 0, BLACK = 1;
}

interface Separates {
    int CYAN = 0, MAGENTA = 1, YELLOW = 2, BLACK = 3;
}

class Test implements Colors, Separates {
    public static void main(String[] args) {
        System.out.println(BLACK); // compile-time error: ambiguous
    }
}
```

main 方法中的名称 BLACK 是有歧义的, 这是由于类 Test 具有两个名为 BLACK 的成员, 一个继承自 Colors, 另一个则继承自 Separates。

如果类类型的方法具有并非重写等价的 (override-equivalent) (8.4.2 节) 签名, 则该类类型可能具有两个或多个具有相同简单名称的方法。这样一个方法成员名称被称为重载。

当一个方法是另外继承自超类或超接口时, 类类型可以包含具有相同名称和相同签名的方法的声明。在这种情况下, 超类或超接口的方法不会被继承。如果未被继承的方法是 abstract 类型, 则称新的声明是实现它; 如果未被继承的方法不是 abstract 类型, 则称新的声明是重写它。

在下面的示例中:

```
class Point {
    float x, y;
    void move(int dx, int dy) { x += dx; y += dy; }
    void move(float dx, float dy) { x += dx; y += dy; }
    public String toString() { return "("+x+","+y+")"; }
}
```

类 Point 具有两个成员, 它们是同名方法 move。为任何特殊方法调用选择的类 Point 的重载 move 方法, 是在编译时通过重载第 15.12 节中给出的求解过程确定的。

在本示例中, 类 Point 的成员有: Point 中声明的 float 实例变量 x 和 y、两个声明的 move 方法、声明的 toString 方法, 以及 Point 从其隐式直接超类 Object (4.3.2 节) 继承的成员, 如方法 hashCode。注意, Point 不会继承类 Object 的 toString 方法, 这是由于该方法会被类 Point 中 toString 方法的声明重写。

### 6.4.4 接口类型的成员

接口类型（9.2 节）的成员可能是类（8.5 节、9.5 节）、接口（8.5 节、9.5 节）、字段（8.3 节、9.3 节、10.7 节）和方法（8.4 节、9.4 节）。接口的成员有：

- 接口中声明的那些成员。
- 从直接超接口继承的那些成员。
- 如果一个接口没有直接超接口，那么该接口将隐式声明一个公共抽象成员方法 *m*，该方法具有签名 *s*、返回类型 *r* 和 throws 子句 *t*，对应于 Object 中声明的具有签名 *s*、返回类型 *r* 和 throws 子句 *t* 的每个公共实例方法 *m*，除非该接口显式声明了一个具有相同签名、相同返回类型和兼容的 throws 子句的方法。如果该接口显式声明了这样一个方法 *m*，而 Object 中则把 *m* 声明成 final 类型，那么就会发生编译时错误。

类型变量（4.4 节）不是成员。

如果字段是在不同接口中声明的或者继承自不同的接口，则一个接口可能具有两个或多个具有相同简单名称的字段。尝试通过其简单名称指代任何一个这样的字段都会导致编译时错误（6.5.6.1 节、9.2 节）。

在下面的示例中：

```
interface Colors {
    int WHITE = 0, BLACK = 1;
}
interface Separates {
    int CYAN = 0, MAGENTA = 1, YELLOW = 2, BLACK = 3;
}
interface ColorsAndSeparates extends Colors, Separates {
    int DEFAULT = BLACK; // compile-time error: ambiguous
}
```

接口 ColorsAndSeparates 的成员包括从 Colors 和 Separates 继承的成员，即 WHITE、BLACK（两个 BLACK 中的前一个）、CYAN、MAGENTA、YELLOW 和 BLACK（两个 BLACK 中的后一个）。在接口 ColorsAndSeparates 中，成员名称 BLACK 是有歧义的。

### 6.4.5 数组类型的成员

第 10.7 节中指定了数组类型的成员。为方便起见，我们在这里重复介绍了该规范。

数组类型的成员包括下列所有成员：

- public final 字段 length，它包含数组元素的个数（length 可能为正数或 0）。
- public 方法 clone，它重写了 Object 类中的同名方法，并且不会抛出未经检查的异常。数组类型 *T*[] 的 clone 方法的返回类型是 *T*[]。
- 从 Object 类继承的所有成员；未被继承的 Object 的惟一方法是它的 clone 方法。

示例：

```
class Test {
    public static void main(String[] args) {
```

```
int[] ia = new int[3];
int[] ib = new int[6];
System.out.println(ia.getClass() == ib.getClass());
System.out.println("ia has length=" + ia.length);
}
```

产生如下输出：

```
true
ia has length=3
```

本示例使用从 `Object` 类继承的方法 `getClass` 和字段 `length`。在第一个 `println` 中，`Class` 对象的比较结果证实，其元素为 `int` 类型的所有数组都是相同数组类型（即 `int[]`）的实例。

## 6.5 确定名称的含义

名称的含义取决于使用它的环境。确定名称的含义需要三个步骤。首先，环境从语法上导致一个名称属于以下 6 个类别之一：`PackageName`、`TypeName`、`ExpressionName`、`MethodName`、`PackageOrTypeName` 或 `AmbiguousName`。其次，最初被其环境分类为 `AmbiguousName` 或 `PackageOrTypeName` 的名称随后会被重新分类为 `PackageName`、`TypeName` 或 `ExpressionName`。第三，随后得到的类别规定名称含义的最终确定（如果名称没有含义，则会发生编译错误）。

*PackageName:*

*Identifier*

*PackageName . Identifier*

*TypeName:*

*Identifier*

*PackageOrTypeName . Identifier*

*ExpressionName:*

*Identifier*

*AmbiguousName . Identifier*

*MethodName:*

*Identifier*

*AmbiguousName . Identifier*

*PackageOrTypeName:*

*Identifier*

*PackageOrTypeName . Identifier*

*AmbiguousName:*



*Identifier**AmbiguousName . Identifier*

使用环境有助于把不同种类的实体之间的名称冲突减至最少。如果遵循第 6.8 节中描述的命名约定，则这种冲突将很少见。然而，当由不同程序员或不同组织开发的类型演变时，这些冲突就有可能不经意地发生。例如，类型、方法和字段可能具有相同的名称。由于使用的环境总是会告知一个方法是否是想要的，因此总是有可能区分具有相同名称的方法和字段。

### 6.5.1 依据环境从语法上对名称分类

在下面这些环境中，从语法上把名称分类为 *PackageName*：

- 在包声明（7.4 节）中
- 在限定性 *PackageName* 中的 “.” 的左边

在下面这些环境中，从语法上把名称分类为 *TypeName*：

- 在单一类型导入声明（7.5.1 节）中
- 在单一静态导入（7.5.3 节）声明中的 “.” 的左边
- 在按需静态导入（7.5.4 节）声明中的 “.” 的左边
- 在参数化类型（4.5 节）中的 “<” 的左边
- 在参数化类型的实际类型参数列表中
- 在泛型方法（8.4.4 节）或构造函数（8.8.4 节）调用中的显式实际类型参数列表中
- 在类型变量声明（8.1.2 节）中的 *extends* 子句中
- 在通配符类型参数（4.5.1 节）的 *extends* 子句中
- 在通配符类型参数（4.5.1 节）的 *super* 子句中
- 在类声明中的 *extends* 子句中（8.1.4 节）
- 在类声明中的 *implements* 子句中（8.1.5 节）
- 在接口声明中的 *extends* 子句中（9.1.3 节）
- 在注释（9.7 节）中的 “@” 符号后面
- 作为下列任何环境中的类型（或者在所有括号被删除后会保持下来的类型的一部分）：
  - ◆ 在字段声明（8.3 节、9.3 节）中
  - ◆ 作为方法（8.4 节、9.4 节）的结果类型
  - ◆ 作为方法或构造函数的形参的类型（8.4.1 节、8.8.1 节、9.4 节）
  - ◆ 作为可以被方法或构造函数抛出的异常的类型（8.4.6 节、8.8.5 节、9.4 节）
  - ◆ 作为局部变量（14.4 节）的类型
  - ◆ 作为 *try* 语句（14.20 节）的 *catch* 子句中异常参数的类型
  - ◆ 作为类字面常数（15.8.2 节）中的类型
  - ◆ 作为限定性 *this* 表达式（15.8.4 节）的合格类型
  - ◆ 作为类类型，它将在非限定性类实例创建表达式（15.9 节）中进行实例化



- ◆ 作为匿名类（15.9.5 节）的直接超类或直接超接口，该匿名类将在非限定性类实例创建表达式（15.9 节）中进行实例化
- ◆ 作为将在数组创建表达式（15.10 节）中创建的数组的元素类型
- ◆ 作为使用关键字 `super` 的字段访问（15.11.2 节）的合格类型
- ◆ 作为使用关键字 `super` 的方法调用（15.12 节）的合格类型
- ◆ 作为强制转换表达式（15.16 节）的强制转换运算符中提及的类型
- ◆ 作为遵循 `instanceof` 关系运算符（15.20.2 节）的类型

在下面这些环境中，从语法上把名称分类为 *ExpressionName*：

- 作为限定性超类构造函数调用（8.8.7.1 节）中的合格表达式
- 作为限定性类实例创建表达式（15.9 节）中的合格表达式
- 作为数组访问表达式（15.13 节）中的数组引用表达式
- 作为 *PostfixExpression*（15.14 节）
- 作为赋值运算符（15.26 节）的左操作数

在下面这些环境中，从语法上把名称分类为 *MethodName*：

- 在方法调用表达式（15.12 节）中的“(”之前
- 在注释的元素值对（9.7 节）中的“=”符号的左边

在下面这些环境中，从语法上把名称分类 *PackageOrTypeName* 为：

- 在限定性 *TypeName* 中的“.”的左边
- 在按需类型导入声明（7.5.2 节）中

在下面这些环境中，从语法上把名称分类 *AmbiguousName* 为：

- 在限定性 *ExpressionName* 中的“.”的左边
- 在限定性 *MethodName* 中的“.”的左边
- 在限定性 *AmbiguousName* 中的“.”的左边
- 在注释类型元素声明（9.6 节）的默认值子句中
- 在元素值对（9.7 节）中的“=”符号的右边

## 6.5.2 重新分类环境中有歧义的名称

然后对 *AmbiguousName* 进行重新分类，如下：

- ◆ 如果 *AmbiguousName* 是简单名称，包含单一 *Identifier*：
  - ◆ 如果 *Identifier* 出现在具有该名称的局部变量声明（14.4 节）、参数声明（8.4.1 节、8.8.1 节、14.20 节）或字段声明（8.3 节）的作用域（6.3 节）内，那么就将 *AmbiguousName* 重新分类为 *ExpressionName*。
  - ◆ 否则，如果通过单一静态导入声明（7.5.3 节）或者通过按需静态导入声明（7.5.4 节）在包含 *Identifier* 的编译单元（7.3 节）中声明具有该名称的字段，那么就将 *AmbiguousName* 重新分类为 *ExpressionName*。
  - ◆ 否则，如果 *Identifier* 出现在具有该名称的顶级类（第 8 章）或接口类型声明（第 9 章）、本地类声明（14.3 节）或成员类型声明（8.5 节、9.5 节）的作用域内（6.3

节), 那么就将 *AmbiguousName* 重新分类为 *TypeName*。

- ◆ 否则, 如果通过单一类型导入声明 (7.5.1 节)、按需类型导入声明 (7.5.2 节)、单一静态导入声明 (7.5.3 节) 或者通过按需静态导入声明 (7.5.4 节) 在包含 *Identifier* 的编译单元 (7.3 节) 中声明具有该名称的类型, 那么就将 *AmbiguousName* 重新分类为 *TypeName*。
- ◆ 否则, 将 *AmbiguousName* 重新分类为 *PackageName*。后一步确定是否实际存在具有该名称的包。
- 如果 *AmbiguousName* 是限定名称, 包含名称、“.” 和 *Identifier*, 那么将首先重新分类“.”左边的名称, 因为它自身就是 *AmbiguousName*。然后可以选择:
  - ◆ 如果把“.”左边的名称重新分类为 *PackageName*, 则若有一个包, 它的名称是“.”左边的名称, 并且该包包含其名称与 *Identifier* 相同的类型的声明, 那么就会将这个 *AmbiguousName* 重新分类为 *TypeName*。否则, 就会将这个 *AmbiguousName* 重新分类为 *PackageName*。后一步将确定是否实际存在具有该名称的包。
  - ◆ 如果把“.”左边的名称重新分类为 *TypeName*, 那么如果 *Identifier* 是 *TypeName* 指定的类型的方法或字段的名称, 则会将这个 *AmbiguousName* 重新分类为 *ExpressionName*。否则, 如果 *Identifier* 是 *TypeName* 指定的类型的成员类型的名称, 则会将这个 *AmbiguousName* 重新分类为 *TypeName*; 否则, 就会导致编译时错误。
  - ◆ 如果把“.”左边的名称重新分类为 *ExpressionName*, 那么设 *T* 是 *ExpressionName* 指定的表达式的类型。如果 *Identifier* 是 *T* 指定的类型的方法或字段的名称, 则会将这个 *AmbiguousName* 重新分类为 *ExpressionName*。否则, 如果 *Identifier* 是 *T* 指定的类型的成员类型 (8.5 节、9.5 节) 的名称, 则会将这个 *AmbiguousName* 重新分类为 *TypeName*; 否则, 就会导致编译时错误。

在下面的示例中, 考虑以下精心设计的“库代码”:

```
package org.rpgpoet;
import java.util.Random;
interface Music { Random[] wizards = new Random[4]; }
```

然后考虑另一个包中的这段示例代码:

```
package bazola;
class Gabriel {
    static int n = org.rpgpoet.Music.wizards.length;
}
```

首先, 会将名称 `org.rpgpoet.Music.wizards.length` 重新分类为 *ExpressionName*, 这是因为它用作一个 *PostfixExpression*。因此, 下列每个名称:

```
org.rpgpoet.Music.wizards
org.rpgpoet.Music
org.rpgpoet
org
```

最初都被分类为 *AmbiguousName*。然后对这些名称重新进行分类:

- 简单名称 `org` 被重新分类为 *PackageName* (这是由于作用域中没有名为 `org` 的变量或类型)。
- 接下来, 假定有包 `org` 的任何编译单元中没有名为 `rpgpoet` 的类或接口 (并且我们知道由于包 `org` 具有名为 `rpgpoet` 的子包, 因此没有这样的类或接口), 则将限定名称 `org.rpgpoet` 重新分类为 *PackageName*。
- 接下来, 由于包 `org.rpgpoet` 具有名为 `Music` 的接口类型, 则将限定名称 `org.rpgpoet.Music` 重新分类为 *TypeName*。
- 最后, 由于名称 `org.rpgpoet.Music` 是一个 *TypeName*, 则将限定名称 `org.rpgpoet.Music.wizards` 重新分类为 *ExpressionName*。

### 6.5.3 包名称的含义

分类为 *PackageName* 的名称含义的确定方式如下。

#### 6.5.3.1 简单的包名称

如果包名称包含单一标识符, 那么这个标识符就指定了通过该标识符命名的顶级包。如果用作域 (7.4.4 节) 中没有具有该名称的顶级包, 则会发生编译时错误。

#### 6.5.3.2 限定性包名称

如果包名称是 `Q.Id` 形式, 那么 `Q` 也必须是一个包名称。包名称 `Q.Id` 命名了一个包, 它是名为 `Q` 的包内名为 `Id` 的成员。如果 `Q` 没有命名一个可观察的包 (7.4.3 节), 或者 `Id` 不是那个包的可观察子包的简单名称, 那么就会发生编译时错误。

### 6.5.4 PackageOrTypeNames 的含义

#### 6.5.4.1 简单的 PackageOrTypeNames

如果 *PackageOrTypeName* (即 `Q`) 出现在名为 `Q` 的类型的的作用域中, 则会将 *PackageOrTypeName* 重新分类为 *TypeName*。

否则, *PackageOrTypeName* 将被重新分类为 *PackageName*。 *PackageOrTypeName* 的含义就是重新分类的名称的含义。

#### 6.5.4.2 限定性 PackageOrTypeNames

给定 `Q.Id` 形式的限定性 *PackageOrTypeName*, 如果 `Q` 指定的类型或包具有名为 `Id` 的成员类型, 那么就将限定性 *PackageOrTypeName* 名称重新分类为 *TypeName*。

否则, 则将其重新分类为 *PackageName*。限定性 *PackageOrTypeName* 的含义就是重新分类的名称的含义。

### 6.5.5 类型名称的含义

分类为 *TypeName* 的名称含义的确定方式如下。

### 6.5.5.1 简单类型名称

如果一个类型名称包含单一标识符，那么该标识符必须出现在具有该名称的类型的完全可见声明的作用域，否则就会发生编译时错误。类型名称的含义就是那个类型。

### 6.5.5.2 限定性类型名称

如果类型名称是  $Q.Id$  形式，那么  $Q$  必须是一个类型名称或包名称。如果  $Id$  正好命名了一个类型，而该类型是  $Q$  指定的类型或包的成员，则限定性类型名称就指示了那种类型。如果  $Id$  没有命名  $Q$  内的一个成员类型（8.5 节、9.5 节），或者  $Q$  内  $Id$  命名的成员类型不可访问（6.6 节），那么就会发生编译时错误。

下面的示例：

```
package wnj.test;
class Test {
    public static void main(String[] args) {
        java.util.Date date =
            new java.util.Date(System.currentTimeMillis());
        System.out.println(date.toLocaleString());
    }
}
```

在第一次运行时会产生如下输出：

```
Sun Jan 21 22:56:29 1996
```

在本示例中，名称 `java.util.Date` 必须指示一种类型，因此我们首先递归地使用过程来确定 `java.util` 是否是可访问的类型或包，如果是，则设法查看类型 `Date` 在这个包中是否是可访问的。

#### 问题

类型名称不同于类型声明指定符（4.3 节）。类型名称总是通过另一个类型名称限定的。在某些情况下，有必要访问作为参数化类型的成员的内部类：

```
class GenericOuter<T extends Number> {
    public class Inner<S extends Comparable<S>> {
        T getT() { return null;}
        S getS() { return null;}
    }
};
GenericOuter<Integer>.Inner<Double> x1 = null;
Integer i = x1.getT();
Double d = x1.getS();
```

如果我们通过用类型名称限定 `Inner` 来访问它，例如：

```
GenericOuter.Inner x2 = null;
```

我们就强制它用作原生类型，从而丢失了类型信息。

## 6.5.6 表达式名称的含义

分类为 *ExpressionName* 的名称含义的确定方式如下。

### 6.5.6.1 简单的表达式名称

如果一个表达式名称包含单一标识符，那么在该标识符必须出现的那一点，作用域中必须正好有一个可见声明指示局部变量、参数或字段。否则，就会发生编译时错误。

如果该声明声明了一个 *final* 字段，则名称的含义就是该字段的值。否则，表达式名称的含义就是该声明所声明的变量。

如果字段是一个实例变量（8.3 节），那么表达式名称必须出现在实例方法（8.4 节）、构造函数（8.8 节）、实例初始化语句（8.6 节）或实例变量初始化语句（8.3.2.2 节）的声明内。如果它出现在 *static* 方法（8.4.3.2 节）、静态初始化语句（8.7 节）或静态变量的初始化语句（8.3.2.1 节、12.4.2 节）内，则会发生编译时错误。

表达式名称的类型是捕获转换（5.1.10 节）之后的字段、局部变量或参数的声明类型。在下面的示例中：

```
class Test {
    static int v;
    static final int f = 3;
    public static void main(String[] args) {
        int i;
        i = 1;
        v = 2;
        f = 33; // compile-time error
        System.out.println(i + " * " + v + " * " + f);
    }
}
```

用作 *i*、*v* 和 *f* 的赋值语句左端项的名称指示局部变量 *i*、字段 *v* 和 *f* 的值（不是变量 *f*，因为 *f* 是一个 *final* 变量）。因此，本示例将在编译时产生一个错误，因为最后一条赋值语句不会具有作为其左端项的变量。如果删除错误的赋值语句，就可以编译修改过的代码，它将产生如下输出：

```
1 2 3
```

### 6.5.6.2 限定性表达式名称

如果表达式名称是 *Q.Id* 形式，则 *Q* 已经被分类为一个包名称、类型名称或表达式名称：

- 如果 *Q* 是包名称，那么就会发生编译时错误。
- 如果 *Q* 是命名类类型（第 8 章）的类型名称，那么：
  - ◆ 若正好没有一个类类型的可访问的（6.6 节）成员，它是名为 *Id* 的字段，那么就会发生编译时错误。
  - ◆ 否则，如果单一可访问的成员字段不是类变量（即它未被声明为 *static*），那么就会发生编译时错误。



- ◆ 否则，如果类变量被声明为 `final`，那么 `Q.Id` 指示类变量的值。表达式 `Q.Id` 的类型是捕获转换（5.1.10 节）之后类变量的声明类型。如果 `Q.Id` 出现在需要一个变量而不是一个值的环境中，那么就会发生编译时错误。
- ◆ 否则，`Q.Id` 指示类变量。表达式 `Q.Id` 的类型是捕获转换（5.1.10 节）之后类变量的声明类型。注意，这个子句涉及使用枚举常量（8.9 节），因为它们总是具有对应的 `final` 类变量。
- 如果 `Q` 是命名接口类型（第 9 章）的类型名称，那么：
  - ◆ 若正好没有一个接口类型的可访问的（6.6 节）成员，它是名为 `Id` 的字段，那么就会发生编译时错误。
  - ◆ 否则，`Q.Id` 指示字段的值。表达式 `Q.Id` 的类型是捕获转换（5.1.10 节）之后字段的声明类型。如果 `Q.Id` 出现在需要一个变量而非一个值的环境中，那么就会发生编译时错误。
- 如果 `Q` 是一个表达式名称，设 `T` 是表达式 `Q` 的类型：
  - ◆ 若 `T` 不是引用类型，则会发生编译时错误。
  - ◆ 若正好没有一个 `T` 类型的可访问的（6.6 节）成员，它是名为 `Id` 的字段，那么就会发生编译时错误。
  - ◆ 否则，如果该字段是以下字段之一：
    - ◇ 接口类型的字段
    - ◇ 类类型的 `final` 字段（它可以类变量或者实例变量）
    - ◇ 数组类型的 `final` 字段 `length`
 那么 `Q.Id` 就指示字段的值。表达式 `Q.Id` 的类型是捕获转换之后（5.1.10 节）字段的声明类型。如果 `Q.Id` 出现在需要一个变量而非一个值的环境中，那么就会发生编译时错误。
  - ◆ 否则，`Q.Id` 就指示一个变量，即类 `T` 的字段 `Id`，它可能是类变量或实例变量。表达式 `Q.Id` 的类型是捕获转换（5.1.10 节）之后字段成员的类型。

下面的示例：

```
class Point {
    int x, y;
    static int nPoints;
}

class Test {
    public static void main(String[] args) {
        int i = 0;
        i.x++; // compile-time error
        Point p = new Point();
        p.nPoints(); // compile-time error
    }
}
```

会遇到两个编译时错误，这是因为 `int` 变量 `i` 没有成员，并且 `nPoints` 不是类 `Point` 的方法。



## 讨论

注意：一般来讲，表达式名称可以通过类型名称（但不能通过类型）来限定。这导致不可能通过参数化类型访问一个类变量。

```
class Foo<T> {
    public static int classVar = 42;
}
Foo<String>.classVar = 91; // illegal
```

相反，可以写成：

```
Foo.classVar = 91;
```

这不会以任何有意义的方式限制语言。类型参数不能用在静态变量的类型中，因此参数化类型的实参永远不会影响静态变量的类型。所以，不会失去表达能力。从技术上讲，上面的类型名称 `Foo` 是原生类型，但是这样使用原生类型是没有害处的，并且不会引发警告。

## 6.5.7 方法名称的含义

*MethodName* 只能出现在方法调用表达式（15.12 节）中，或者作为元素值对（9.7 节）中的元素名称。分类为 *MethodName* 的名称含义的确定方式如下。

### 6.5.7.1 简单方法名称

简单方法名称可以作为元素值对中的元素名称。*ElementValuePair* 中的 *Identifier* 必须是包含的注释中通过 *TypeName* 标识的注释类型的元素之一的简单名称。否则，就会发生编译时错误（换句话说，元素值对中的标识符还必须是通过 *TypeName* 标识的接口中的方法名称）。

否则，简单的方法名称必须出现在方法调用表达式的环境中。在这种情况下，如果方法名称包含单一 *Identifier*，那么 *Identifier* 就是用于方法调用的方法名称。该 *Identifier* 必须命名至少一个可见的（6.3.1 节）方法，该方法在 *Identifier* 出现的那一点位于作用域中，或者是在该 *Identifier* 出现的编译单元内通过单一静态导入声明（7.5.3 节）或按需静态导入声明（7.5.4 节）导入的方法。

有关方法调用表达式中的简单方法名称解释的进一步讨论，参见第 15.12 节。

### 6.5.7.2 限定性方法名称

限定性方法名称只能出现在方法调用表达式的环境中。如果方法名称是 *Q.Id* 形式，那么 *Q* 已经被分类为一个包名称、类型名称或表达式名称。如果 *Q* 是包名称，那么就会发生编译时错误。否则，*Id* 是用于方法调用的方法名称。如果 *Q* 是类型名称，那么 *Id* 必须命名至少一个 *Q* 类型的 *static* 方法。如果 *Q* 是表达式名称，则设 *T* 是表达式 *Q* 的类型：*Id* 必须命名至少一个 *T* 类型的方法。有关方法调用表达式中的限定性方法名称解释的进一步讨论，参见第 15.12 节。

## 讨论

与表达式名称一样，方法名称通常可以通过类型名称（但是不能通过类型）来限定。其含意类似于第 6.5.6.2 节中讨论的表达式名称的含意。

## 6.6 访问控制

Java 编程语言提供了用于访问控制的机制，以防止包或类的用户依赖于包或类的不必要的实现细节。如果允许访问，就称被访问的实体是可访问的。

注意，可访问性是一个可以在编译时确定的静态属性；它只依赖于类型和声明修饰符。限定性名称是访问包和引用类型的成员的手段；访问的相关手段包括字段访问表达式（15.11 节）和方法调用表达式（15.12 节）。所有这三种访问手段在语法上是类似的，它们都采用一个“.”标记，其前面是具有某种类型的包、类型或表达式的某个指示，其后接有指定包或类型的成员的标识符。这些泛指限定性访问的构造。

通过类实例创建表达式（15.9 节）和显式构造函数调用（8.8.7.1 节），将访问控制应用于限定性访问和构造函数的调用。可访问性还影响了类成员（8.2 节）的继承，包括隐藏和方法重写（8.4.8.1 节）。

### 6.6.1 确定可访问性

- 包始终是可访问的。
- 如果类或接口类型被声明为 `public`，那么它可以被任何代码访问，假定在其中声明它的编译单元（7.3 节）是可观察的。如果顶级类或接口类型未被声明为 `public`，那么只能从声明它们的包内部访问它们。
- 当且仅当数组的元素类型可访问时，数组类型才是可访问的。
- 当且仅当类型是可访问的并且成员或构造函数被声明为允许访问时，引用（类、接口或数组）类型的成员（类、接口、字段或方法）或类类型的构造函数才是可访问的：
  - ◆ 如果成员或构造函数被声明为 `public`，那么就允许访问。接口的所有成员隐式都是 `public`。
  - ◆ 否则，如果成员或构造函数被声明为 `protected`，那么仅当下列条件之一成立时，才允许访问：
    - ◇ 从包内部访问成员或构造函数，该包包含在其中声明 `protected` 成员或构造函数的类。
    - ◇ 访问是正确的，如第 6.6.2 节中所述。
  - ◆ 否则，如果成员或构造函数被声明为 `private`，那么当且仅当访问发生在顶级类（7.6 节）的主体内，并且该顶级类封闭了成员或构造函数的声明时，才允许访问。
  - ◆ 否则，我们称存在默认访问，当且仅当从声明类型的包内部进行访问时，才允许进行这种默认访问。

## 6.6.2 关于 protected 访问的详细信息

如果要从声明对象的包外部访问该对象的 `protected` 成员或构造函数，则只能通过负责实现该对象的代码来进行。

### 6.6.2.1 访问 protected 成员

设  $C$  是在其中声明 `protected` 成员的类。只允许在  $C$  的子类  $S$  的主体内部进行访问。此外，如果  $Id$  指示一个实例字段或实例方法，那么：

- 若通过限定名称  $Q.Id$  进行访问，其中  $Q$  是一个 *ExpressionName*，那么当且仅当表达式  $Q$  的类型是  $S$  或  $S$  的子类时，才允许进行访问。
- 若通过字段访问表达式  $E.Id$  进行访问，其中  $E$  是一个主表达式，或者通过方法调用表达式  $E.Id(\dots)$  进行访问，其中  $E$  是一个主表达式，那么当且仅当  $E$  的类型是  $S$  或  $S$  的子类时，才允许进行访问。

### 6.6.2.2 对 protected 构造函数的限定性访问

设  $C$  是在其中声明 `protected` 构造函数的类， $S$  是在其声明中使用 `protected` 构造函数的最内部的类。那么：

- 若通过超类构造函数调用 `super(\dots)` 或者通过  $E.super(\dots)$  形式的限定性超类构造函数调用进行访问，其中  $E$  是主表达式，那么允许进行访问。
- 若通过 `new C(\dots)\{\dots\}` 形式的匿名类实例创建表达式或者通过  $E.new C(\dots)\{\dots\}$  形式的限定性类实例创建表达式进行访问，其中  $E$  是主表达式，那么允许进行访问。
- 否则，如果通过 `new C(\dots)` 形式的简单类实例创建表达式，或者通过  $E.new C(\dots)$  形式的限定性类实例创建表达式进行访问，其中  $E$  是主表达式，那么不允许进行访问。只能从定义它的包内部通过类实例创建表达式（它没有声明匿名类）访问 `protected` 构造函数。

## 6.6.3 访问控制的示例

对于下面的访问控制的示例，考虑两个编译单元：

```
package points;
class PointVec { Point[] vec; }
```

和：

```
package points;
public class Point {
    protected int x, y;
    public void move(int dx, int dy) { x += dx; y += dy; }
    public int getX() { return x; }
    public int getY() { return y; }
}
```

它们在包 `points` 中声明了两个类类型：

- 类类型 `PointVec` 不是 `public`，并且不是包 `points` 的 `public` 接口的一部分，只能被该包内的其他类使用。
- 类类型 `Point` 被声明为 `public`，并且可供其他的包使用。它是包 `points` 的 `public` 接口的一部分。
- 类 `Point` 的方法 `move`、`getX` 和 `getY` 被声明为 `public`，因此可供使用类型 `Point` 的对象的任何代码使用。
- 字段 `x` 和 `y` 被声明为 `protected`，仅当它们是被正在访问它们的代码实现的对象的字段时，才只能在类 `Point` 的子类中从包 `points` 的外部访问这两个字段。

有关 `protected` 访问修饰符如何限制访问的示例，参见第 6.6.7 节。

#### 6.6.4 示例：访问 `public` 和非 `public` 类

如果类缺少 `public` 修饰符，则对类声明的访问仅限于在其中声明类的包（6.6 节）。在下面的示例中：

```
package points;
public class Point {
    public int x, y;
    public void move(int dx, int dy) { x += dx; y += dy; }
}
class PointList {
    Point next, prev;
}
```

在编译单元中声明了两个类。类 `Point` 可以从包 `points` 外部访问，而类 `PointList` 则只能从包内部访问。

因此，另一个包中的编译单元可以访问 `points.Point`，这可以通过使用其完全限定的名称来进行：

```
package pointsUser;
class Test {
    public static void main(String[] args) {
        points.Point p = new points.Point();
        System.out.println(p.x + " " + p.y);
    }
}
```

或者通过使用提及完全限定名称的简单类型导入声明（7.5.1 节）来进行，因而，此后可以使用简单名称：

```
package pointsUser;
import points.Point;
class Test {
    public static void main(String[] args) {
        Point p = new Point();
    }
}
```

```
System.out.println(p.x + " " + p.y);  
}}
```

但是，这个编译单元不能使用或导入 `points.PointList`，它未被声明为 `public`，因此不能从包 `points` 外部访问它。

### 6.6.5 示例：默认访问字段、方法和构造函数

如果没有指定访问修饰符 `public`、`protected` 或 `private` 中的任何一个，则可以在整个包内访问类成员或构造函数，如果这个包包含在其中声明类成员的类的声明，但是，不能在任何其他包中访问该类成员或构造函数。

如果 `public` 类具有默认访问的方法或构造函数，那么该方法或构造函数不能被在该包外部声明的子类访问，也不能被其继承。

例如，如果我们具有：

```
package points;  
public class Point {  
    public int x, y;  
    void move(int dx, int dy) { x += dx; y += dy; }  
    public void moveAlso(int dx, int dy) { move(dx, dy); }  
}
```

那么另一个包中的子类可以声明一个不相关的 `move` 方法，它具有相同的签名（8.4.2 节）和返回类型。由于不能从包 `morepoints` 访问原始的 `move` 方法，因此可以不使用 `super`：

```
package morepoints;  
public class PlusPoint extends points.Point {  
    public void move(int dx, int dy) {  
        super.move(dx, dy); // compile-time error  
        moveAlso(dx, dy);  
    }  
}
```

由于 `Point` 的 `move` 不会被 `PlusPoint` 中的 `move` 重写，`Point` 中的方法 `moveAlso` 永远不会调用 `PlusPoint` 中的方法 `move`。

因此，如果从 `PlusPoint` 删除 `super.move` 调用，并执行下列测试程序：

```
import points.Point;  
import morepoints.PlusPoint;  
class Test {  
    public static void main(String[] args) {  
        PlusPoint pp = new PlusPoint();  
        pp.move(1, 1);  
    }  
}
```

它会正常终止。如果 `Point` 的 `move` 被 `PlusPoint` 中的 `move` 重写，那么本程序将无限递归，直到发生 `StackOverflowError`。



### 6.6.6 示例：public 字段、方法和构造函数

在声明它们的整个包中以及从其他任何包中都可以访问 public 类成员或构造函数，假定在其中声明它们的包是可观察的（7.4.3 节）。例如，在下面的编译单元中：

```
package points;
public class Point {
    int x, y;
    public void move(int dx, int dy) {
        x += dx; y += dy;
        moves++;
    }
    public static int moves = 0;
}
```

public 类 Point 具有 public 成员，即 move 方法和 moves 字段。这些 public 成员对于可以访问包 points 的其他任何包都是可访问的。字段 x 和 y 不是 public，因此只能从包 points 内部访问。

### 6.6.7 示例：protected 字段、方法和构造函数

考虑下面这个示例，其中 points 包声明了：

```
package points;
public class Point {
    protected int x, y;
    void warp(threePoint.Point3d a) {
        if (a.z > 0) // compile-time error: cannot access a.z
            a.delta(this);
    }
}
```

threePoint 包声明了：

```
package threePoint;
import points.Point;
public class Point3d extends Point {
    protected int z;
    public void delta(Point p) {
        p.x += this.x; // compile-time error: cannot access p.x
        p.y += this.y; // compile-time error: cannot access p.y
    }
    public void delta3d(Point3d q) {
        q.x += this.x;
        q.y += this.y;
        q.z += this.z;
    }
}
```



它定义了一个类 `Point3d`。这里在 `delta` 方法中会发生编译时错误：它不能访问其参数 `p` 的 `protected` 成员 `x` 和 `y`，这是由于虽然 `Point3d`（在该类中发生了对字段 `x` 和 `y` 的引用）是 `Point`（在该类中声明了 `x` 和 `y`）的子类，但是它未包括在 `Point`（参数 `p` 的类型）的实现中。方法 `delta3d` 可以访问其参数 `q` 的 `protected` 成员，这是由于类 `Point3d` 是 `Point` 的子类，并且包括在 `Point3d` 的实现中。

方法 `delta` 可能尝试将其参数强制转换（5.5 节、15.16 节）成 `Point3d`，但是如果 `p` 的类在运行时不是 `Point3d`，这种强制转换将会失败，并引发一个异常。

在方法 `warp` 中也会发生一个编译时错误：它不能访问其参数 `a` 的 `protected` 成员 `z`，这是由于虽然类 `Point`（在该类中发生了对字段 `z` 的引用）包括在 `Point3d`（参数 `a` 的类型）的实现中，但是它不是 `Point3d`（在该类中声明了 `z`）的子类。

## 6.6.8 示例：private 字段、方法和构造函数

`private` 类成员或构造函数只能在封闭了该成员或构造函数声明的顶级类（7.6 节）的主体内部进行访问。它不会被子类继承。在下面的示例中：

```
class Point {
    Point() { setMasterID(); }
    int x, y;
    private int ID;
    private static int masterID = 0;
    private void setMasterID() { ID = masterID++; }
}
```

`private` 成员 `ID`、`masterID` 和 `setMasterID` 只能在类 `Point` 的主体内部使用。它们不能被位于 `Point` 声明主体外部的限定名称、字段访问表达式或方法调用表达式访问。

有关使用 `private` 构造函数的示例，参见第 8.8.8 节。

## 6.7 完全限定名称和规范名称

每个包、顶级类、顶级接口和基本类型都有一个完全限定名称。当且仅当数组的元素具有完全限定名称时，这个数组类型才有完全限定名称。

- 基本类型的完全限定名称是该基本类型的关键字，即 `boolean`、`char`、`byte`、`short`、`int`、`long`、`float` 或 `double`。
- 如果指定包不是另一个指定包的子包，则该指定包的完全限定名称是其简单名称。
- 如果指定包是另一个指定包的子包，则该指定包的完全限定名称包含所包含包的完全限定名称，后接“.”，其后再接子包的简单（成员）名称。
- 在未指定包中声明的顶级类或顶级接口的完全限定名称是该类或接口的简单名称。
- 在未指定包中声明的顶级类或顶级接口的完全限定名称包含包的完全限定名称，后接“.”，其后再接该类或接口的简单名称。
- 当且仅当另一个类 `C` 具有完全限定的名称时，`C` 的成员类或成员接口 `M` 才具有完全

限定的名称。在这种情况下，*M*的完全限定名称包含*C*的完全限定名称，后接“.”，其后再接*M*的简单名称。

- 数组类型的完全限定名称包含该数组类型的元素类型的完全限定名称，其后接“[]”。

示例：

- 类型 `long` 的完全限定名称是“`long`”。
- 包 `java.lang` 的完全限定名称是“`java.lang`”，因为它是包 `java` 的子包 `lang`。
- 类 `Object`（在包 `java.lang` 中定义）的完全限定名称是“`java.lang.Object`”。
- 接口 `Enumeration`（在包 `java.util` 中定义）的完全限定名称是“`java.util.Enumeration`”。
- 类型“`double` 数组”的完全限定名称是“`double[]`”。
- “`String` 数组的数组的数组的数组”的完全限定名称是“`java.lang.String[][][]`”。

在下面的示例中：

```
package points;
class Point { int x, y; }
class PointVec {
    Point[] vec;
}
```

类型 `Point` 的完全限定名称是“`points.Point`”；类型 `PointVec` 的完全限定名称是“`points.PointVec`”；类 `PointVec` 的字段 `vec` 的类型的完全限定名称是“`points.Point[]`”。

每个包、顶级类、顶级接口和基本类型都具有规范名称。当且仅当数组类型的元素类型具有规范名称时，该数组类型才具有规范名称。当且仅当另一个类 *C* 具有规范名称时，*C* 中声明的成员类或成员接口 *M* 才具有规范名称。在这种情况下，*M* 的规范名称包含 *C* 的规范名称，后接“.”，其后再接 *M* 的简单名称。对于每个包、顶级类、顶级接口和基本类型，规范名称与完全限定名称相同。当且仅当数组的元素类型具有规范名称时，才会定义数组类型的规范名称。在这种情况下，数组类型的规范名称包含该数组类型的元素类型的规范名称，后接“[]”。

可以在如下所示的示例中看到完全限定名称和规范名称之间的区别：

```
package p;
class O1 { class I{} }
class O2 extends O1{};
```

在本示例中，`p.O1.I` 和 `p.O2.I` 都是指示同一个类的完全限定名称，但是只有 `p.O1.I` 是其规范名称。

## 6.8 命名约定

只要有可能，Java 平台的类库就会尝试使用依据这里介绍的约定而选择的名称。这些约定有助于使代码更可读，并且避免某些类型的名称冲突。

我们建议在有 Java 编程语言编写的所有程序中使用这些约定。但是，如果长期保持的

传统用法另有不同的规定，则不应该盲目遵循这些约定。因此，例如，类 `java.lang.Math` 的 `sin` 和 `cos` 方法在数字上具有约定的名称，即使这些方法名称由于比较短并且不是动词而忽视了这里建议的约定。

### 6.8.1 包名称

为了使包的名称广泛可用，应该按第 7.7 节中的描述创建它们。这样的名称总是限定名称，它的第一个标识符包含两个或三个小写字母，它们指定了一个 Internet 域，如 `com`、`edu`、`gov`、`mil`、`net`、`org`；或者是两个字母的 ISO 国家代码，如 `uk` 或 `jp`。下面是在该约定下可能创建的假定惟一名称的示例：

```
com.JavaSoft.jag.Oak
org.npr.pledge.driver
uk.ac.city.rugby.game
```

仅打算供本地使用的包的名称应该具有以小写字母开头的第一个标识符，但是第一个标识符明确地不应该是标识符 `java`；开始于标识符 `java` 的包名称是 Sun 保留的，用于命名 Java 平台包。

当包名称出现在表达式中时：

- 如果字段声明模糊了包名称，那么 `import` 声明（7.5 节）通常可用于使那个包中声明的类型名称可用。
- 如果参数或局部变量的声明模糊了包名称，那么在不影响其他代码的情况下，可以更改该参数或局部变量的名称。

包名称的第一个组成部分通常不容易被误认为类型名称，因为类型名称通常开始于单个大写字母（Java 编程语言实际上不依赖于大小写区别来确定一个名称是包名称，还是类型名称）。

### 6.8.2 类和接口类型名称

类类型的名称应该是说明性的名词或名词短语，不要使名称过长，采用大小写混合的形式，并且大写每个单词的首字母。例如：

```
ClassLoader
SecurityManager
Thread
Dictionary
BufferedInputStream
```

同样，接口类型的名称应该保持较短且具有说明性，不要使名称过长，采用大小写混合的形式，并且大写每个单词的首字母。该名称应该是说明性的名词或名词短语，当像抽象超类那样使用接口时，这是合适的，如接口 `java.io.DataInput` 和 `java.io.DataOutput`；或者它也可以是描述行为的形容词，如接口 `Runnable` 和 `Cloneable`。

涉及类和接口类型名称的模糊很少见。字段、参数和局部变量的名称通常不会模糊类型名称，这是由于它们传统上以小写字母开头，而类型名称传统上以大写字母开头。

### 6.8.3 类型变量名称

类型变量名称应该简练（如果可能的话，可使用单个字符）而具说明性，并且不应该包括小写字母。

#### 讨论

这使得容易区分形式类型参数与普通类和接口。

容器类型应该为其元素类型使用名称 *E*。映射应该为其键的类型使用 *K*，为其值的类型使用 *V*。名称 *X* 应该用于任意的异常类型。当无需关于某个类型的更多知识即可区分它时，我们都应该为类型使用 *T*。

#### 讨论

这通常是泛型方法中的情况。

如果有多个指示任意类型的类型参数，则其中一个应该使用字母表中 *T* 的邻近字母，如 *S*。一种替代方法是，使用数字下标（例如，*T1*、*T2*）来区分不同的类型变量是可接受的。在这种情况下，具有相同前缀的所有变量都应该带有下标。

#### 讨论

如果泛型方法出现在泛型类的内部，那么一个好的想法是：避免为方法和类的类型参数使用相同的名称，以避免混淆。这同样适用于嵌套的泛型类。

#### 讨论

下面的代码段中阐释了这些约定：

```
public class HashSet<E> extends AbstractSet<E> { ... }
public class HashMap<K,V> extends AbstractMap<K,V> { ... }
public class ThreadLocal<T> { ... }
public interface Functor<T, X extends Throwable> {
    T eval() throws X;
}
```

当不能方便地把类型参数归入提及的类别之一时，应该在单个字母的限制内，尽可能地选择有意义的名称。不应该为不属于指定类别的类型参数使用上面提及的名称（*E*、*K*、*T*、*V*、*X*）。

### 6.8.4 方法名称

方法名称应该是动词或动词短语，采用大小写混合的形式，名称的首字母小写，而任



何后续单词的首字母则大写。这里介绍了方法名称的一些额外的具体约定：

- 应该把用于 `get` 和 `set` 一个可能被视为变量 *V* 的属性的方法命名为 `getV` 和 `setV`。例如，类 `Thread` 的方法 `getPriority` 和 `setPriority`。
- 应该把返回某个对象长度的方法命名为 `length`，例如，在类 `String` 中。
- 对于测试关于某个对象的 `boolean` 条件 *V* 的方法，应该将其命名为 `isV`。例如，类 `Thread` 的方法 `isInterrupted`。
- 对于将其对象转换成一种特殊格式 *F* 的方法，应该将其命名为 `toF`。例如，类 `Object` 的方法 `toString` 以及类 `java.util.Date` 的方法 `toLocaleString` 和 `toGMTString`。

无论何时可能并且合适，使新类中的方法名称基于类似的现有类（特别是 Java 应用程序编程接口类中的类）中的名称，将使得其易于使用。

方法名称不能模糊其他名称，反之亦然（6.5.7 节）。

### 6.8.5 字段名称

不是 `final` 的字段名称应该采用大小写混合的形式，它的首字母要小写，而后续每个单词的首字母则大写。注意：除了常量字段（`final static` 字段）（6.8.6 节）外，良好设计的类极少有 `public` 或 `protected` 字段。

字段的名称应该是名词、名词短语或名词的简写。这种约定的示例有：类 `java.io.ByteArrayInputStream` 的字段 `buf`、`pos` 和 `count`，以及类 `java.io.InterruptedIOException` 的字段 `bytesTransferred`。

涉及字段名称的模糊很少见。

- 如果字段名称模糊了包名称，那么通常可使用 `import` 声明（7.5 节）来使那个包中声明的类型名称可用。
- 如果字段名称模糊了类型名称，那么可以使用该类型的完全限定名称，除非该类型名称指示一个本地类（14.3 节）。
- 字段名称不能模糊方法名称。
- 如果字段名称被参数或局部变量的声明所屏蔽，那么在不影响其他代码的情况下，可以更改参数或局部变量的名称。

### 6.8.6 常量名称

接口类型中的常量名称应该是一个或多个单词、缩写词或简写词的序列，它们全都大写，各个成分之间用下划线“`_`”字符隔开，类类型的 `final` 变量传统上也可以采用这样的命名方式。常量名称应该具有说明性，不应该不必要地进行简写。传统上讲，它们可以是语言的任何合适的部分。常量名称的示例包括：类 `Character` 的 `MIN_VALUE`、`MAX_VALUE`、`MIN_RADIX` 和 `MAX_RADIX`。

通常情况下，有时用公共缩写词作为名称前缀，来指定用于表示集合的可选值或者（较少见）整型值中的屏蔽位的一组常量，如下：

```
interface ProcessStates {  
    int PS_RUNNING = 0;  
    int PS_SUSPENDED = 1;  
}
```

涉及常量名称的模糊很少见：

- 常量名称中通常没有小写字母，因此它们通常不会模糊包或类型的名称，也不会屏蔽字段，字段的名称中通常包含至少一个小写字母。
- 常量名称不能模糊方法名称，因为它们在语法上是有区别的。

### 6.8.7 局部变量和参数名称

局部变量和参数应该比较短，但有意义。它们通常是小写字母（而不是单词）的较短的序列。例如：

- 一系列单词的首字母缩写词，如 `cp`，用于保存指向 `ColoredPoint` 的引用的变量
- 简写词，如 `buf`，用于保存指向某种 `buffer` 的指针
- 助记词，通常通过使用一组具有约定名称的局部变量以某种方式组织它们，以帮助记忆和理解，这些名称作为模板接在广泛使用的类的参数名称后面。例如：
  - ◆ `in` 和 `out`，无论何时涉及某种类型的输入和输出，它们都会作为模板接在 `System` 的字段后面
  - ◆ `off` 和 `len`，无论何时涉及偏移量和长度，它们都会作为模板接在 `java.io` 的接口 `DataInput` 和 `DataOutput` 的 `read` 和 `write` 方法的参数后面

除了临时变量和循环变量，或者保存类型的不可区分值的变量之外，应该避免使用单字符局部变量或参数名称。约定的单字符名称有：

- `b`，用于 `byte`
- `c`，用于 `char`
- `d`，用于 `double`
- `e`，用于 `Exception`
- `f`，用于 `float`
- `i`、`j` 和 `k`，用于整数
- `l`，用于 `long`
- `o`，用于 `Object`
- `s`，用于 `String`
- `v`，用于某种类型的任意值

只包含两个或三个小写字母的局部变量或参数不应该与初始国家代码和域名称[它们是惟一的包名称（7.7节）的第一个成分]发生冲突。



名称有什么关系呢？玫瑰即使不叫玫瑰，依然芳香如故。  
——威廉·莎士比亚，《罗密欧和朱丽叶》第二幕第二场

玫瑰就是玫瑰，就是玫瑰，就是玫瑰。  
——Gertrude Stein, 《Sacred Emily》(1913), 选自《Geographies and Plays》

第一朵玫瑰的名字揭示了一切。  
——莫雷的贝尔纳，《De contemptu mundi》(12世纪), 摘自《玫瑰的名字》(1980)

*Rose, Rose, bo-Bose,  
Banana-fana fo-Fose,  
Fee, fie, mo-Mose—  
—Rose!*

——Lincoln Chase 和 Shirley Elliston, 摘自《The Name Game》，适合于名称“Rose”

# 第7章

## 包

乡人不识货，尽捡大的摸。  
——传统谚语

程序被组织成包的集合。每个包都具有它自己的类型名称集，这有助于防止名称冲突。仅当某个顶级类型被声明为 `public` 时，该类型在声明它的包外部才是可访问的（6.6 节）。

包的命名结构是分层的（7.1 节）。包的成员是类和接口类型（7.6 节），它们是在包的编译单元和子包中声明的，这些子包可能包含编译单元以及它们自己的子包。

包可以存储在文件系统（7.2.1 节）或数据库（7.2.2 节）中。存储在文件系统中的包可能具有关于其编译单元组织方式的某些约束条件，以允许简单实现轻松地查找类。

包包含许多编译单元（7.3 节）。编译单元自动具有对其包中声明的所有类型的访问权限，并且还会自动导入预定义的包 `java.lang` 中声明的所有公共类型。

对于小型程序和临时的开发，包可以不命名（7.4.2 节）或者具有简单名称，但是若要广泛分发代码，则应该选择惟一的包名称（7.7 节）。如果两个开发组碰巧选择了相同的包名称，并且稍后将把这些包用在单独一个程序中，那么这可以防止发生冲突。

### 7.1 包成员

包的成员有：它的子包，以及包的所有编译单元（7.3 节）中声明的所有顶级（7.6 节）类类型（第 8 章）和顶级接口类型（第 9 章）。

例如，在 Java 应用程序编程接口中：

- 包 `java` 具有子包 `awt`、`applet`、`io`、`lang`、`net` 和 `util`，但是没有编译单元。
- 包 `java.awt` 具有名为 `image` 的子包，以及包含类和接口类型声明的许多编译单元。

如果包的完全限定名称（6.7 节）是 `P`，`Q` 是 `P` 的子包，那么 `P.Q` 就是该子包的完全限定名称。

包不能包含两个同名的成员，否则会导致编译时错误。

下面是一些示例：

- 由于包 `java.awt` 具有子包 `image`，它不能（并且不会）包含名为 `image` 的类或

接口类型的声明。

- 如果有一个名为 `mouse` 的包，以及该包中的一个成员类型 `Button`（这样可以称之为 `mouse.Button`），那么就不能有任何具有完全限定名称 `mouse.Button` 或 `mouse.Button.Click` 的包。
- 如果 `com.sun.java.jag` 是类型的完全限定名称，那么就不能有其完全限定名称为 `com.sun.java.jag` 或 `com.sun.java.jag.scrabble` 的任何包。

包的分层命名结构旨在以一种约定的方式方便地组织相关的包，但是，这种命名结构除了禁止一个包拥有在该包中声明的作为顶级类型（7.6 节）的具有相同简单名称的子包外，它自身并没有重要价值。在名为 `oliver` 的包和另一个名为 `oliver.twist` 的包之间，或者在名为 `evelyn.wood` 的包和名为 `evelyn.waugh` 的包之间并没有特殊的访问关系。例如，与任何其他包中的代码相比，名为 `oliver.twist` 的包中的代码并不具有对 `oliver` 包内声明的类型更好的访问权限。

## 7.2 包的主机支持

每台主机确定了包、编译单元和子包的创建和存储方式，以及特殊编译中的哪些编译单元是可观察的（7.3 节）。

编译单元的可观察性反过来又确定了哪些包是可观察的，以及哪些包在作用域中。

包可以存储在 Java 平台的简单实现的本地文件系统中。其他实现可以使用分发的文件系统或者某种形式的数据库，来存储源代码和/或二进制代码。

### 7.2.1 在文件系统中存储包

作为一个极其简单的示例，系统上所有的包以及源代码和二进制代码可以存储在单个目录及其子目录中。该目录的每个直接子目录将代表一个顶级包，也就是说，其完全限定名称包含单一的简单名称。目录可能包含以下直接子目录：

```
com
gls
jag
java
wnj
```

其中，目录 `java` 包含 Java 应用程序编程接口包；目录 `jag`、`gls` 和 `wnj` 可能包含本规范的三位作者为其个人使用以及与这个小组内的其他成员彼此共享而创建的包；目录 `com` 可能包含从公司获得的包，这些公司使用第 7.7 节中描述的约定为它们的包生成唯一的名称。

继续这个示例，除了别的子目录之外，目录 `java` 还将包含下列子目录：

```
applet
awt
io
lang
```

```
net
util
```

它们对应于包 `java.applet`、`java.awt`、`java.io`、`java.lang`、`java.net` 和 `java.util`，这些包被定义为 Java 应用程序编程接口的一部分。

仍然继续这个示例，如果我们查看目录 `util` 的内部，就可以看到下面这些文件：

```
BitSet.java           Observable.java
BitSet.class          Observable.class
Date.java             Observer.java
Date.class            Observer.class
...
```

其中每个 `.java` 文件都包含编译单元（7.3 节）的源代码，该编译单元包含类或接口的定义，而这些类或接口的二进制编译形式则包含在相应的 `.class` 文件中。

在包的这种简单的组织结构之下，Java 平台的实现将通过串接包名称的各个成分，在邻近的成分之间放置一个文件名分隔符（目录指示符），把包名称转换成一个路径名。

例如，如果把这种简单的组织结构用在 UNIX 系统上，其中文件名分隔符是 `/`，则包名称：

```
jag.scrabble.board
```

将被转换成目录名称：

```
jag/scrabble/board
```

并且：

```
com.sun.sunsoft.DOE
```

将被转换成目录名称：

```
com/sun/sunsoft/DOE
```

包名称组件或类名称可能包含不能正确显示在主机文件系统的普通目录名称中的字符，如 Unicode 字符在只允许在文件名中使用 ASCII 字符的系统上将不能正确显示。作为一个约定，可以通过使用（例如）`@` 字符对一个字符进行转义，`@` 字符后面接 4 个十六进制的数字，用于给那个字符赋值，如 `\uxxxx` 转义（3.3 节），因此对于包名称：

```
children.activities.crafts.papierM\u00e2ch\u00e9
```

也可以使用完全的 Unicode 字符将其写成：

```
children.activities.crafts.papierMâché
```

这可能映射到目录名称：

```
children/activities/crafts/papierMâché
```

如果对于某些给定的主机文件系统，`@` 字符不是文件名中的有效字符，那么可以代之以使用某个在标识符中无效的其他字符。

## 7.2.2 在数据库中存储包

主机系统可以把包及其编译单元和子包存储在数据库中。

这样的数据库绝对禁止在基于文件的实现中的编译单元上强加可选的限制条件（7.6节）。例如，使用数据库存储包的系统可能不会强制每个编译单元最多包含一个 `public` 类或接口。

但是，使用数据库的系统必须提供一个选项，以将程序转换成服从这些限制条件的形式，用于导出到基于文件的实现。

## 7.3 编译单元

*CompilationUnit* 是 Java 程序的语义语法（2.3 节）的目标符（2.1 节），它通过如下产生式定义：

*CompilationUnit*:

*PackageDeclaration<sub>opt</sub> ImportDeclarations<sub>opt</sub> TypeDeclarations<sub>opt</sub>*

*ImportDeclarations*:

*ImportDeclaration*

*ImportDeclarations ImportDeclaration*

*TypeDeclarations*:

*TypeDeclaration*

*TypeDeclarations TypeDeclaration*

在不同编译单元中声明的类型可能循环地相互依赖。Java 编译器必须安排同时编译所有这样的类型。

编译单元包含 3 部分，其中每个部分都是可选的：

- `package` 声明（7.4 节），提供包含该编译单元的包的完全限定名称（6.7 节）。没有包声明的编译单元是未命名包（7.4.2 节）的一部分。
- `import` 声明（7.5 节），允许使用简单名称引用来自其他包的类型以及类型的静态成员。
- 类和接口类型的顶级类型声明（7.6 节）。

至于哪些编译单元是可观察的，这是由主机系统确定的。但是，包 `java` 及其子包 `lang` 和 `io` 的所有编译单元必须总是可观察的。编译单元的可观察性影响了其包的可观察性（7.4.3 节）。

每个编译单元会自动和隐式地导入通过预定义的包 `java.lang` 声明的每个 `public` 类型名称，以使得所有这些类型的名称都可以作为简单名称使用，如第 7.5.5 节中所述。

## 7.4 包声明

包声明出现在编译单元内，用于指示编译单元属于哪个包。

### 7.4.1 命名的包

编译单元中的包声明指定了该编译单元所属的包的名称（6.2 节）。

*PackageDeclaration:*

*Annotations<sub>opt</sub>* package *PackageName* ;

可以可选地把注释修饰符(9.7节)置于关键字 package 的前面, 如果包声明上的注释 *a* 对应于注释类型 *T*, 并且 *T* 具有(元)注释 *m*, 它对应于 annotation.Target, 那么 *m* 必须具有一个值为 annotation.ElementType.PACKAGE 的元素, 否则就会发生编译时错误。

包声明中提及的包名称必须是包的完全限定名称(6.7节)。

## 7.4.1.1 包注释

注释可用在包声明上, 这有一个限制, 即对于给定的包, 至多允许一个加过注释的包声明。

强制执行这种限制的方式不可避免地因实现而异。强烈建议为基于文件系统的实现使用下列模式: 唯一加过注释的包声明(如果它存在的话)被置于称为 package-info.java 的源文件中, 而该文件位于包含那个包的源文件的目录中。该文件不包含称为 package-info.java 的类的源代码; 实际上, 这是非法的, 因为 package-info 不是合法的标识符。通常情况下, package-info.java 只包含一个包声明, 在其前面紧接着关于包的注释。虽然从技术上讲, 该文件可以包含一个或多个 package-private 类的源代码, 但这是一种非常糟糕的形式。

对于 javadoc 及其他类似的文档生成系统, 建议用 package-info.java (如果它存在的话) 代替 package.html。如果该文件存在, 那么文档生成工具应该寻找 package-info.java 中紧接在(可能加过注释的)包声明前面的包文档注释。这样, package-info.java 就成为包级别的、注释和文档惟一的存储库。将来, 如果希望添加任何其他包级别的信息, 该文件应该证明它是该信息的方便的存储位置。

## 7.4.2 未命名的包

没有包声明的编译单元是未命名的包的一部分。

注意, 未命名的包不能有子包, 因为包声明的语法总是包括一个指向命名的顶级包的引用。例如, 下面的编译单元:

```
class FirstCall {
    public static void main(String[] args) {
        System.out.println("Mr. Watson, come here. "
            + "I want you.");
    }
}
```

把一个非常简单的编译单元定义为未命名的包的一部分。

Java 平台的实现必须支持至少一个未命名的包; 它可以支持多个未命名的包, 但是不



需要这样做。每个未命名的包中有哪些编译单元，是由主机系统确定的。

在使用用于存储包的分层文件系统的 Java 平台的实现中，一个典型的策略是把未命名的包与每个目录关联起来：一次只有一个未命名的包是可观察的，即与“当前工作目录”关联的那个包是可观察的。“当前工作目录”的精确含义依赖于主机系统。

在开发小型或临时应用程序时，或者仅仅在开始开发时，未命名的包是由 Java 平台主要出于方便性目的而提供的。

### 7.4.3 包的可观察性

当且仅当下列条件之一成立时包才是可观察的：

- 包含包声明的编译单元是可观察的。
- 包的一个子包是可观察的。

通过上述规则以及可观察的编译单元的要求，可以得出结论：包 `java`、`java.lang` 和 `java.io` 总是可观察的。

### 7.4.4 包声明的作用域

可观察的（7.4.3 节）顶级包声明的作用域是所有可观察的编译单元（7.3 节）。不可观察的包声明永远不在作用域内。子包声明永远不在作用域内。

包 `java` 总是在作用域内（6.3 节）。

包声明永远不会屏蔽其他声明。

## 7.5 导入声明

导入声明允许通过包含单一标识符的简单名称（6.2 节）来引用静态成员或指定的类型。如果不使用合适的 `import` 声明，那么引用在另一个包中声明的类型或者另一个类型的静态成员的惟一方式是使用完全限定名称（6.7 节）。

*ImportDeclaration:*

*SingleTypeImportDeclaration*

*TypeImportOnDemandDeclaration*

*SingleStaticImportDeclaration*

*StaticImportOnDemandDeclaration*

单一类型导入声明（7.5.1 节）通过提及其规范名称（6.7 节），可以导入单一指定的类型。按需类型导入声明（7.5.2 节）可以根据需要导入指定的类型或包的所有可访问的（6.6 节）类型。从未命名的包导入类型将会引发编译时错误。

单一静态导入声明（7.5.3 节）通过提供其规范名称，利用类型的给定名称导入所有可访问的静态成员。

按需静态导入声明（7.5.4 节）可以根据需要导入指定类型的所有可访问的静态成员。

通过单一类型导入声明（7.5.1 节）或按需类型导入声明（7.5.2 节）导入的类型的作

用域是包含导入声明的编译单元中的所有类和接口类型声明（7.6节）。

通过单一静态导入声明（7.5.3节）或按需静态导入声明（7.5.4节）导入的成员的作用域是包含导入声明的编译单元中的所有类和接口类型声明（7.6节）。

仅仅在实际包含 import 声明的编译单元中，import 声明使得类型可以通过其简单名称使用。它引用的实体的作用域明确地不包括 package 语句、当前编译单元中的其他 import 声明，或者相同包中的其他编译单元。参见第 7.5.6 节，以查看一个说明性的示例。

### 7.5.1 单一类型导入声明

单一类型导入声明通过赋予其规范名称，导入单一的类型，从而使得可以通过包含单一类型导入声明的编译单元的类和接口声明中的简单名称来使用它。

*SingleTypeImportDeclaration:*

```
import TypeName;
```

*TypeName* 必须是类或接口类型的规范名称；如果指定的类型不存在，则会发生编译时错误。指定的类型必须是可访问的（6.6节），否则会发生编译时错误。

对于包 *p* 的编译单元 *c* 中的单一类型导入声明 *d*，如果它导入名为 *n* 的类型，则会屏蔽整个 *c* 中的以下声明：

- 另一个编译单元 *p* 中声明的任何名为 *n* 的顶级类型。
- 通过 *c* 中的按需类型导入声明导入的任何名为 *n* 的类型。
- 通过 *c* 中的按需静态导入声明导入的任何名为 *n* 的类型。

示例：

```
import java.util.Vector;
```

导致简单名称 *Vector* 在编译单元中的类和接口类型声明内可用。因此，简单名称 *Vector* 将会引用所有位置的包 *java.util* 中的类型声明 *Vector*，只要它在这些位置没有被同名的字段、参数、局部变量的声明或嵌套的类型声明屏蔽（6.3.1节）或模糊（6.3.2节）即可。



注意，*Vector* 被声明为泛型类型。一旦被导入，无需具有合格的参数化类型，如 *Vector<String>*，即可使用名称 *Vector*，或者将其用作原生类型 *Vector*。

这突出显示了 import 声明的限制。可以导入嵌套在泛型类型声明内部的类型，但是其外部类型总会被擦除。

如果相同编译单元中的两个单一类型导入声明尝试导入具有相同简单名称的类型，那么就会发生编译时错误，除非两个类型是相同的类型，在这种情况下会忽视完全一样的声明。如果通过单一类型导入声明导入的类型是在包含该导入声明的编译单元中声明的，则会忽视导入声明。如果编译单元包含单一静态导入声明（7.5.3节）（用于导入一个简单名

称为  $n$  的类型) 和单一类型导入声明 (7.5.1 节) (用于导入一个简单名称为  $n$  的类型), 则就会发生编译时错误。

如果通过按需类型导入声明 (7.5.2 节) 或按需静态导入声明 (7.5.4 节) 在当前编译单元中另外声明了另一个具有相同简单名称的顶级类型, 就会发生编译时错误。

因此, 下列示例程序:

```
import java.util.Vector;
class Vector { Object[] vec; }
```

由于 `Vector` 的完全一样的声明, 而引发编译时错误, 如下:

```
import java.util.Vector;
import myVector.Vector;
```

其中 `myVector` 是包含编译单元的包:

```
package myVector;
public class Vector { Object[] vec; }
```

编译器通过类型的二进制名称来跟踪它们 (13.1 节)。

注意: 导入语句不能导入子包, 而只能导入类型。例如, 它不会试图导入 `java.util`, 然后使用名称 `util.Random` 引用类型 `java.util.Random`:

```
import java.util; // incorrect: compile-time error
class Test { util.Random generator; }
```

## 7.5.2 按需类型导入声明

按需类型导入声明允许根据需要来导入通过规范名称命名的类型或包中声明的所有可访问的 (6.6 节) 类型。

*TypeImportOnDemandDeclaration:*

```
import PackageOrTypeName.*;
```

如果按需类型导入声明指定一个不可访问的类型或包, 则会发生编译时错误。相同编译单元中的两个或多个按需类型导入声明可以指定相同的类型或包。除了其中一个之外, 其余所有的声明都被认为是冗余的; 其效果就好像是类型只被导入了一次。

如果编译单元同时包含指定了相同类型的按需静态导入声明和按需类型导入声明 (7.5.2 节), 其效果就好像是该类型的静态成员类型只被导入了一次。

在按需类型导入声明中指定当前的包或 `java.lang` 不会引发编译时错误。在这种情况下, 会忽视按需类型导入声明。

按需类型导入声明永远不会导致任何其他的声明被屏蔽。

示例:

```
import java.util.*;
```

导致包 `java.util` 中声明的所有 `public` 类型的名称在编译单元的和接口声明中都可用。因此, 简单名称 `Vector` 会引用编译单元中所有位置的包 `java.util` 中的类型 `Vector`, 在该编译单元中, 那种类型声明不会被屏蔽 (6.3.1 节) 或模糊 (6.3.2 节)。该

声明可能会被其简单名称为 `Vector` 的类型的单一类型导入声明所屏蔽；或者被包含该编译单元的包中声明的名为 `Vector` 的类型所屏蔽；抑或被任何嵌套的类或接口所屏蔽。该声明可能被名为 `Vector` 的字段、参数或局部变量的声明模糊（这些情况都很少见）。

### 7.5.3 单一静态导入声明

单一静态导入声明通过类型的给定简单名称导入所有可访问的（6.6 节）静态成员。这使得在包含该单一静态导入声明的编译单元的和接口声明中可以通过其简单名称来使用这些静态成员。

*SingleStaticImportDeclaration:*

```
import static TypeName . Identifier;
```

*TypeName* 必须是类或接口类型的规范名称；如果指定的类型不存在，则会发生编译时错误。指定的类型必须是可访问的（6.6 节），否则会发生编译时错误。*Identifier* 必须指定所指定类型的至少一个静态成员；如果没有那个名称的成员或者如果所有指定的成员都不可访问，则会发生编译时错误。

对于包 *p* 的编译单元 *c* 中的单一静态导入声明 *d*，如果它导入一个名为 *n* 的字段，则会在整个 *c* 中屏蔽 *c* 中通过按需静态导入声明导入的任何名为 *n* 的静态字段的声明。

对于包 *p* 的编译单元 *c* 中的单一静态导入声明 *d*，如果它导入一个名为 *n* 的具有签名 *s* 的方法，则会在整个 *c* 中屏蔽 *c* 中通过按需静态导入声明导入的任何名为 *n* 的具有签名 *s* 的静态方法的声明。

对于包 *p* 的编译单元 *c* 中的单一静态导入声明 *d*，如果它导入一个名为 *n* 的类型，则会在整个 *c* 中屏蔽以下声明：

- 在 *c* 中通过按需静态导入声明导入的任何名为 *n* 的静态类型。
- *p* 的另一个编译单元（7.3 节）中声明的任何名为 *n* 的顶级类型（7.6 节）。
- 在 *c* 中通过按需类型导入声明（7.5.2 节）导入的任何名为 *n* 的类型。

注意，允许一个单一静态导入声明导入多个具有相同名称的字段或类型，或者多个具有相同名称和签名的方法。

如果编译单元包含单一静态导入声明（7.5.3 节）（用于导入一个简单名称为 *n* 的类型）和单一类型导入声明（7.5.1 节）（用于导入一个简单名称为 *n* 的类型），那么就会发生编译时错误。

如果单一静态导入声明导入了一个简单名称为 *n* 的类型，并且编译单元还声明了一个简单名称为 *n* 的顶级类型（7.6 节），那么就会发生编译时错误。

### 7.5.4 按需静态导入声明

按需静态导入声明允许根据需要来导入通过规范名称命名的类型中声明的所有可访问的（6.6 节）静态成员。

*StaticImportOnDemandDeclaration:*

```
import static TypeName . *;
```



如果按需静态导入声明指定一个不存在的类型或者不可访问的类型，则会发生编译时错误。相同编译单元中的两个或多个按需静态导入声明可以指定相同的类型或包，其效果就好像是刚好只有一个这样的声明。相同编译单元中的两个或多个按需静态导入声明可以指定同一个成员，其效果就好像是该成员刚好只被导入一次。

注意，允许一个按需静态导入声明导入多个具有相同名称的字段或类型，或者多个具有相同名称和签名的方法。

如果编译单元同时包含指定了相同类型的按需静态导入声明和按需类型导入声明（7.5.2 节），其效果就好像是该类型的静态成员类型只被导入了一次。

按需静态导入声明永远不会导致任何其他声明被屏蔽。

### 7.5.5 自动导入

每个编译单元都会自动导入预定义的包 `java.lang` 中声明的所有 `public` 类型名称，如同出现在每个编译单元开始处的声明：

```
import java.lang.*;
```

其后紧接着任何 `package` 语句。

### 7.5.6 一个奇怪的示例

在第 6.8 节中描述的命名约定之下，包名称和类型名称通常不同。然而，在一个精心设计的示例中，其中有一个不按照约定命名的包 `Vector`，它声明了一个 `public` 类，其名称是 `Mosquito`：

```
package Vector;
public class Mosquito { int capacity; }
```

则编译单元如下：

```
package strange.example;
import java.util.Vector;
import Vector.Mosquito;
class Test {
    public static void main(String[] args) {
        System.out.println(new Vector().getClass());
        System.out.println(new Mosquito().getClass());
    }
}
```

从包 `java.util` 中导入类 `Vector` 的单一类型导入声明（7.5.1 节）不会阻止出现包名称 `Vector`，也不会阻止它在后续 `import` 声明中被正确地识别。该示例通过编译后会产生以下输出：

```
class java.util.Vector
class Vector.Mosquito
```

## 7.6 顶级类型声明

顶级类型声明声明了一个顶级类类型（第8章）或一个顶级接口类型（第9章）：

*TypeDeclaration:*

*ClassDeclaration*

*InterfaceDeclaration*

;

默认情况下，包中声明的顶级类型只能在该包的编译单元内部访问，但是可以把一个类型声明为 `public`，以授权从其他包中代码访问该类型（6.6节、8.1.1节、9.1.1节）。

顶级类型的作用域是声明顶级类型的包中的所有类型声明。

如果在完全限定名称为  $P$  的包的编译单元中声明一个名为  $T$  的顶级类型，那么该类型的完全限定名称就是  $P.T$ 。如果在未命名的包（7.4.2节）中声明该类型，则该类型的完全限定名称为  $T$ 。

因此，在下面的示例中：

```
package wnj.points;
class Point { int x, y; }
```

类 `Point` 的完全限定名称是 `wnj.points.Point`。

Java 平台的实现必须通过类型的二进制名称（13.1节）来跟踪包内的类型。必须将命名类型的多种方式扩展到二进制名称，以确保这种名称被理解为引用相同的类型。

例如，如果编译单元包含单一类型导入声明（7.5.1节）：

```
import java.util.Vector;
```

那么在该编译单元内，简单名称 `Vector` 和完全限定名称 `java.util.Vector` 引用相同的类型。

当把包存储在文件系统中时（7.2.1节），如果以下两个条件之一成立，那么如果在组合类型名称以及一个扩展名（如 `.java` 或 `.jav`）作为文件名称的文件中没有找到一个类型，则主机系统可以选择强制执行一种限制，即这是一个编译时错误：

- 通过声明类型的包的其他编译单元中的代码引用类型。
- 类型被声明为 `public`（因此，潜在地可以被其他包中的代码访问）。

这种限制暗示每个编译单元必须至多包含一个这样的类型。这种限制使得 Java 编程语言的编译器或 Java 虚拟机的实现可以轻松地找到包内指定的类：例如，可以在目录 `wet/sprocket` 中找到 `public` 类型 `wet.sprocket.Toad` 的源代码，其对应的目标代码则可以在同一目录中的 `Toad.class` 文件找到。

当把包存储在数据库中时（7.2.2节），主机系统绝对禁止强加这种限制。在实践中，许多程序员选择把每个类或接口类型放在它自己的编译单元中，而不管它是否是公共类型，或者它是否被其他编译单元中的代码引用。

如果某个顶级类型的名称作为同一个包中声明的任何其他顶级类或接口类型的名称出现，那么就会发生编译时错误（7.6节）。



如果通过包含类型声明的编译单元（7.3 节）中的单一类型导入声明（7.5.1 节）把顶级类型的名称还声明为一种类型，那么就会发生编译时错误。

在下面的示例中：

```
class Point { int x, y; }
```

类 Point 是在不含 package 语句的编译单元中声明的，因此 Point 是它的完全限定名称，而在下面的示例中：

```
package vista;
class Point { int x, y; }
```

类 Point 的完全限定名称是 vista.Point [包名称 vista 适合于局部或个人使用；如果打算广泛分发这个包，更好的方式是赋予其惟一的包名称（7.7 节）]。

在下面的示例中：

```
package test;
import java.util.Vector;
class Point {
    int x, y;
}
interface Point { // compile-time error #1
    int getR();
    int getTheta();
}
class Vector { Point[] pts; } // compile-time error #2
```

第一个编译时错误是由把名称 Point 作为同一个包中的 class 和 interface 的完全一样的声明引起的。在编译时检测到的第二个错误是：试图同时通过类类型声明和单一类型导入声明来声明名称 Vector。

但是要注意，用类的名称来命名可能另外通过包含类声明的编译单元（7.3 节）中的按需类型导入声明（7.5.2 节）导入的类型，这不是一个错误。在下面的示例中：

```
package test;
import java.util.*;
class Vector { Point[] pts; } // not a compile-time error
```

允许声明类 Vector，即使还有一个类 java.util.Vector。在这个编译单元内，简单名称 Vector 引用类 test.Vector，而不是引用 java.util.Vector (java.util.Vector 仍然可以被编译单元内的代码引用，但只能通过其完全限定名称来引用)。

作为另一个示例，下面的编译单元：

```
package points;
class Point {
    int x, y; // coordinates
    PointColor color; // color of this point
    Point next; // next point with this color
    static int nPoints;
}
class PointColor {
```

```
Point first; // first point with this color
PointColor(int color) {
    this.color = color;
}
private int color; // color components
}
```

定义了两个类，它们在声明中相互使用对方的类成员。由于类类型 `Point` 和 `PointColor` 把包 `points` 中所有的类型声明（包括当前编译单元中的所有类型声明）作为它们的作用域，所以这个示例将正确地编译——也就是说，向前引用（forward reference）不是一个问题。

如果顶级类型声明包含以下访问修饰符中的任何一个：`protected`、`private` 或 `static`，则会发生编译时错误。

## 7.7 惟一的包名称

我曾告诉你 McCave 夫人有 23 个儿子，并且她把他们都取名为“Dave”。

确实，她这样做了，这是一件明智的事情吗？

——Seuss 博士（Theodore Geisel），《Too Many Daves》（1961）

开发人员应该采取措施，通过为广泛分发的包选择惟一的包名称来避免两个发布的包具有相同名称的可能性。这允许轻松、自动地安装包以及对其编目录。本节详细说明了对于生成这种惟一包名称所建议的约定。Java 平台的实现是被鼓励的，以自动支持把一组包从局部和临时包名称转换成这里描述的惟一名称格式。

如果没有使用惟一的包名称，那么远至创建有冲突的包之一那一刻就可能发生包名称冲突。这可能产生用户或程序员难以或不可能解决的情形。当包具有受约束的交互时，可以使用类 `ClassLoader` 来隔离具有相同名称的包，但是它对于新手编写的程序不是透明的。

可以通过首先具有（或属于一个组织）一个 Internet 域名（如 `sun.com`），来构建惟一的包名称。然后可以逐成分地逆转该名称来获得（在本示例中）`com.sun`，并把它用作包名称的前缀，从而使用你的组织内开发的约定来进一步管理包名称。

在某些情况下，Internet 域名可能不是有效的包名称，下面是处理这些情形的一些推荐的约定：

- 如果域名包含一个连字符，或者标识符（3.8 节）中不允许的任何其他特殊字符，就将其转换成一个下划线字符。
- 如果得到的任何包名称成分是关键字（3.9 节），则为其追加下划线。
- 如果得到的任何包名称成分开始于一个数字，或者不允许作为标识符初始字符的任何其他字符，就对该成分加上下划线前缀。

这种约定可能把某些目录名称成分指定为分公司、部门、项目、机器或登录名称。下面是一些可能的示例：

```
com.sun.sunsoft.DOE
com.sun.java.jag.scrabble
```

```
com.apple.quicktime.v2  
edu.cmu.cs.bovik.cheese  
gov.whitehouse.socks.mousefinder
```

惟一包名称的第一个成分总是用全部小写的 ASCII 字母编写的，并且应该是顶级域名（目前有 com、edu、gov、mil、net、org）之一，或者是 ISO 标准 3166（1981 年）中指定的标识国家的英文两字母代码之一。相关更多信息请参阅 <ftp://rs.internic.net/rfc> 上存储的文档，例如，[rfc920.txt](#) 和 [rfc1032.txt](#)。

包的名称并不打算暗示包存储在 Internet 内的某个位置；例如，不一定能够从 Internet 地址 [cmu.edu](#) 或 [cs.cmu.edu](#) 或 [bovik.cs.cmu.edu](#) 获得名为 [edu.cmu.cs.bovik.cheese](#) 的包。为生成惟一的包名称所建议的约定仅仅只是在现有的、广为人知的惟一名称注册库的顶部附上一个包命名约定，而不是为包名称创建一个单独的注册库。

棕色纸包裹绑上细绳子，这些都是我喜爱的东西。  
——Oscar Hammerstein II, 《My Favorite Things》(1959)

# 第8章

## 类

类 1. 名词 class 派生自中世纪法语，法语 classe 派生自拉丁语 classis，可能最初源于传唤，因此是一个被传唤的人员集合，一个应受传唤的小组：也许是因为响应 calare 的 callassis 的号召，因此被传唤。

——Eric Partridge, 《Origins: A Short Etymological Dictionary of Modern English》

类声明定义了新的引用类型，并描述了它们是如何实现的（8.1 节）。

嵌套类是其声明出现在另一个类或接口的主体内的任何类。顶级类是并非嵌套类的类。

本章讨论了所有类——顶级类（7.6 节）和嵌套类 [包括成员类（8.5 节、9.5 节）、本地类（14.3 节）和匿名类（15.9.5 节）] 的公共语义。在专门针对这些构造的节中讨论了特定于特殊类型的类的详细信息。

命名的类可以被声明为 abstract（8.1.1.1 节），并且如果它没有被完全实现，则必须声明为 abstract；这种类不能被实例化，但是可以被子类扩展。类可以被声明为 final（8.1.1.2 节），在这种情况下，它不能具有子类。如果类被声明为 public，那么可以从其他包中引用它。除 Object 之外的每个类都是单一现有类的一个扩展（即一个子类）（8.1.4 节），并且可以实现接口（8.1.5 节）。类可以是泛型（generic），也就是说，它们可以声明类型变量（4.4 节），它们的绑定在类的不同实例之间可能有所不同。

类可以通过注释（9.7 节）进行修饰，就像任何其他类型的声明一样。

类体声明了成员（字段和方法以及嵌套的类和接口）、实例和静态初始化语句，以及构造函数（8.1.6 节）。成员（8.2 节）的作用域（6.3 节）是该成员所属于的类声明的整个主体。字段、方法、成员类、成员接口和构造函数声明可能包括访问修饰符（6.6 节）public、protected 或 private。类的成员包括声明的和继承的成员（8.2 节）。最新声明的字段可以隐藏超类或超接口中声明的字段。最新声明的类成员和接口成员可以隐藏超类或超接口中声明的类或接口成员。最新声明的方法可以隐藏、实现或重写超类或超接口中声明的方法。

字段声明（8.3 节）描述了类变量（它们只被具体化一次）和实例变量（它们会为类的每个实例都进行最新的具体化）。字段可以被声明为 final（8.3.1.2 节），在这种情况下，它们只能被赋值一次。任何字段声明都可以包括一条初始化语句。

成员类声明（8.5 节）描述了嵌套类，它们是周围的类的成员。成员类可以是 static，

在这种情况下，它们不能访问周围类的实例变量；或者它们是内部类（8.1.3 节）。

成员接口声明（8.5 节）描述了嵌套的接口，它们是周围的类的成员。

方法声明（8.4 节）描述了可以被方法调用表达式（15.12 节）调用的代码。类方法相对于类类型被调用；实例方法相对于某个特殊的对象被调用，该对象是某个类类型的一个实例。其声明未指定其实现方式的方法必须被声明为 `abstract`。方法可以被声明为 `final`（8.4.3.3 节），在这种情况下，它可以被隐藏或重写。方法可以被平台相关的本机代码（8.4.3.4 节）实现。`synchronized` 方法（8.4.3.6 节）在执行其方法体前会自动锁定一个对象，并在返回时自动解除锁定该对象，就像使用 `synchronized` 语句（14.19 节）一样，从而允许其活动与其他线程（第 17 章）的活动进行同步。

方法名称可以被重载（8.4.9 节）。

实例初始化语句（8.6 节）是可执行代码块，它可用于在创建实例时帮助对其进行初始化（15.9 节）。

静态初始化语句（8.7 节）是可执行代码块，它可用于帮助初始化一个类。

构造函数（8.8 节）类似于方法，但是不能直接通过方法调用来调用；它们用于初始化新的类实例。与方法一样，它们可以被重载（8.8.8 节）。

## 8.1 类声明

类声明指定了一个新命名的引用类型。类声明有两种——普通类声明和枚举声明：

*ClassDeclaration:*

*NormalClassDeclaration*

*EnumDeclaration*

*NormalClassDeclaration:*

*ClassModifiers*<sub>opt</sub> **class** *Identifier* *TypeParameters*<sub>opt</sub> *Super*<sub>opt</sub>

*Interfaces*<sub>opt</sub> *ClassBody*

本节中的规则适应于所有的类声明，除非本规范显式指出了其他的情况。在许多情况下，会对枚举声明应用特殊的限制。枚举声明在第 8.9 节中进行了详细描述。

类声明中的 *Identifier* 指定了类的名称。如果类具有与其封闭的任何类或接口相同的简单名称，则会发生编译时错误。

### 8.1.1 类修饰符

类声明可以包括类修饰符。

*ClassModifiers:*

*ClassModifier*

*ClassModifiers* *ClassModifier*

*ClassModifier:* one of



```
Annotation public protected private
abstract static final strictfp
```

并非所有的修饰符都适用于各种类型的类声明。访问修饰符 `public` 仅适合于顶级类 (7.6 节) 和成员类 (8.5 节、9.5 节)，在第 6.6 节、8.5 节和 9.5 节中对此进行了讨论。访问修饰符 `protected` 和 `private` 仅适合于直接封闭的类声明 (8.5 节) 内的成员类，在第 8.5.1 节中对此进行了讨论。访问修饰符 `static` 仅适合于成员类 (8.5 节、9.5 节)。如果相同的修饰符在一个类声明中出现了不止一次，则会发生编译时错误。

如果类声明上的注释 `a` 对应于一个注释类型 `T`，并且 `T` 具有一个 (元) 注释 `m`，它对应于 `annotation.Target`，那么 `m` 必须具有一个元素，其值是 `annotation.ElementType.TYPE`，否则就会发生编译时错误。第 9.7 节中进一步描述了注释修饰符。

如果类声明中出现了两个或多个类修饰符，那么习惯上让它们出现的顺序与上面显示的 `ClassModifier` 的产生式中的顺序一致，尽管不需要如此。

#### 8.1.1.1 abstract 类

`abstract` 类是一个不完整的类，或者被认为是不完整的类。仅当普通类是 `abstract` 类时，它们才可以具有 `abstract` 方法 (8.4.3.1 节、9.4 节)，这些方法是声明但未实现的方法。如果普通类不是 `abstract` 类，但是它包含有 `abstract` 方法，则会发生编译时错误。

枚举类型 (8.9 节) 绝对不能被声明为 `abstract`；否则将导致编译时错误。如果枚举类型 `E` 具有一个 `abstract` 方法 `m` 作为成员，那么除非 `E` 具有一个或多个枚举常量，并且 `E` 的所有枚举常量都具有提供了 `m` 的具体实现的类体，否则就会发生编译时错误。如果枚举常量的类体声明了一个 `abstract` 方法，则会发生编译时错误。

如果以下条件之一成立，则类 `C` 就具有 `abstract` 方法：

- `C` 显式包含 `abstract` 方法 (8.4.3 节) 的声明。
- `C` 的任何超类都具有 `abstract` 方法，并且 `C` 既没有声明也没有继承一个实现 (8.4.8.1 节) 它的方法。
- `C` 的任何超接口 (8.1.5 节) 声明或继承了一个方法 (因此该方法必须是 `abstract`)，并且 `C` 既没有声明也没有继承一个实现它的方法。

在下面的示例中：

```
abstract class Point {
    int x = 1, y = 1;
    void move(int dx, int dy) {
        x += dx;
        y += dy;
        alert();
    }
    abstract void alert();
}
abstract class ColoredPoint extends Point {
    int color;
}
class SimplePoint extends Point {
```

```
void alert() { }  
}
```

类 `Point` 必须被声明为 `abstract`，因为它包含一个名为 `alert` 的 `abstract` 方法的声明。名为 `ColoredPoint` 的 `Point` 的子类继承了 `abstract` 方法 `alert`，因此它也必须被声明为 `abstract`。另一方面，名为 `SimplePoint` 的 `Point` 的子类提供了 `alert` 的实现，因此它不必是 `abstract`。

如果试图使用类实例创建表达式（15.9 节）来创建 `abstract` 类的一个实例，则会发生编译时错误。

因此，如下所示继续这个示例，语句：

```
Point p = new Point();
```

将导致一个编译时错误：类 `Point` 不能被实例化，因为它是 `abstract`。但是，可以通过一个指向 `Point` 的任何子类的引用正确地初始化 `Point` 变量，并且类 `SimplePoint` 不是 `abstract`，因此语句：

```
Point p = new SimplePoint();
```

是正确的。

对于 `abstract` 类的子类，如果它自身不是 `abstract`，则可对其进行实例化，这会导致执行该 `abstract` 类的构造函数，从而执行该类的实例变量的字段初始化语句。因此，在刚才给出的示例中，`SimplePoint` 的实例化将会导致执行 `Point` 的默认构造函数以及 `x` 和 `y` 的字段初始化语句。

如果声明一个 `abstract` 类类型，并且不能创建一个子类来实现它的所有 `abstract` 方法，则会发生编译时错误。如果类具有两个 `abstract` 方法作为成员，这两个方法具有相同的方法签名（8.4.2 节），但是其返回类型不兼容，则会出现这种情形。

例如，以下声明：

```
interface Colorable { void setColor(int color); }  
abstract class Colored implements Colorable {  
    abstract int setColor(int color);  
}
```

将会导致一个编译时错误：类 `Colored` 的任何子类都不可能提供名为 `setColor` 的方法的实现，从而获取一个 `int` 类型的参数，它可以同时满足两种 `abstract` 方法规范，这是由于接口 `Colorable` 中的一个方法需要相同的方法不返回值，而类 `Colored` 中的一个方法则需要相同的方法返回一个 `int` 类型的值（8.4 节）。

仅当意图是可以创建子类来完成实现时，才应该把类类型声明为 `abstract`。如果意图只是阻止类的实例化，那么表达这个意图的合适方式是声明一个无参构造函数（8.8.10 节），使之为 `private` 类型，从而永远不会调用它，并且不声明任何其他的构造函数。这种形式的类通常包含类方法和变量。类 `Math` 是不能被实例化的类的示例：其声明如下：

```
public final class Math {  
    private Math() { } // never instantiate this class  
    ... declarations of class variables and methods ...  
}
```

### 8.1.1.2 final 类

如果类的定义是完整的,并且不希望有或者不需要子类,则可以将该类声明为 `final`。如果 `final` 类的名称出现在另一个类声明的 `extends` 子句(8.1.4 节)中,则会发生编译时错误;这意味着 `final` 类不能具有任何子类。如果类同时被声明为 `final` 和 `abstract`,则会发生编译时错误,因为这样一个类的实现永远不会被完成(8.1.1.1 节)。

由于 `final` 类永远不会有任何子类,因此 `final` 类的方法永远不会被重写(8.4.8.1 节)。

### 8.1.1.3 strictfp 类

`strictfp` 修饰符的作用是使类声明内的所有 `float` 或 `double` 表达式显式地是精确浮点的(15.4 节)。这意味着类中声明的所有方法以及类中声明的所有嵌套类型隐式地是 `strictfp`。

另外请注意:类的所有变量初始化语句、实例初始化语句、静态初始化语句以及构造函数也显式地是精确浮点的。

## 8.1.2 泛型类和类型参数

如果类声明了一个或多个类型变量(4.4 节),则它是一个泛型类。这些类型变量被称为类的类型参数。类型参数部分接着类名,并且通过尖括号来定界。它定义了一个或多个用作参数的类型变量。泛型类声明定义了一组参数化类型,它们用于类型参数部分的每个可能的调用。所有这些参数化类型在运行时共享同一个类。



例如,执行以下代码:

```
Vector<String> x = new Vector<String>();  
Vector<Integer> y = new Vector<Integer>();  
boolean b = x.getClass() == y.getClass();
```

将导致变量 `b` 存储值 `true`。

---

```
TypeParameters ::= < TypeParameterList >  
TypeParameterList ::= TypeParameterList , TypeParameter  
                    | TypeParameter
```

如果泛型类是 `Throwable` 的直接或间接子类,则会发生编译时错误。



由于 Java 虚拟机的捕获机制仅适用于非泛型类,因此需要这种限制。

类的类型参数的作用域是类的整个声明,包括类型参数部分本身。因此,类型参数可以作为它们自身界限的一部分出现,或者作为同一部分中声明的其他类型参数的界限的一

部分出现。

在 *C* 的静态成员的声明中或者在嵌套于 *C* 内的任何类型声明的静态成员的声明中，在任何位置引用类 *C* 的类型参数都会引发一个编译时错误。在 *C* 的静态初始化语句内部或者在嵌套于 *C* 内的任何类内部引用类 *C* 的类型参数将会引发一个编译时错误。

### 讨论

示例：互相递归的类型变量界限。

```
interface ConvertibleTo<T> {
    T convert();
}
class ReprChange<T implements ConvertibleTo<S>,
                S implements ConvertibleTo<T>> {
    T t;
    void set(S s) { t = s.convert(); }
    S get() { return t.convert(); }
}
```

参数化类声明可以嵌套在其他声明的内部。

### 讨论

下面的示例中说明了这一点：

```
class Seq<T> {
    T head;
    Seq<T> tail;
    Seq() { this(null, null); }
    boolean isEmpty() { return tail == null; }
    Seq(T head, Seq<T> tail) { this.head = head; this.tail = tail; }
    class Zipper<S> {
        Seq<Pair<T,S>> zip(Seq<S> that) {
            if (this.isEmpty() || that.isEmpty())
                return new Seq<Pair<T,S>>();
            else
                return new Seq<Pair<T,S>>({
                    new Pair<T,S>(this.head, that.head),
                    this.tail.zip(that.tail);
                })
        }
    }
}
class Pair<T, S> {
    T fst; S snd;
    Pair(T f, S s) { fst = f; snd = s; }
}
class Client {
```

```

    {
        Seq<String> strs =
            new Seq<String>("a", new Seq<String>("b",
                new Seq<String>()));
        Seq<Number> nums =
            new Seq<Number>(new Integer(1),
                new Seq<Number>(new Double(1.5),
                    new Seq<Number>()));
        Seq<String>.Zipper<Number> zipper =
            strs.new Zipper<Number>();
        Seq<Pair<String,Number>> combined = zipper.zip(nums);
    }
}

```

### 8.1.3 内部类和封闭实例

内部类是一个嵌套类，它不能被显式或隐式地声明为 `static`。内部类不能声明静态初始化语句（8.7 节）或成员接口。内部类不能声明静态成员，除非它们是编译时常量字段（15.28 节）。

为了阐释这些规则，考虑下面的示例：

```

class HasStatic{
    static int j = 100;
}
class Outer{
    class Inner extends HasStatic{
        static final int x = 3;    // ok - compile-time constant
        static int y = 4;          // compile-time error, an inner class
    }
    static class NestedButNotInner{
        static int z = 5;          // ok, not an inner class
    }
    interface NeverInner{}        // interfaces are never inner
}

```

内部类可以继承不是编译时常量的静态成员，即使它们不能声明这些成员。依据 Java 编程语言的惯常规则，不是内部类的嵌套类可以自由地声明静态成员。成员接口（8.5 节）隐式地总是静态的，因此它们永远不会被视作内部类。

当且仅当封闭语句或表达式的最内层的方法、构造函数、实例初始化语句、静态初始化语句、字段初始化语句或显式构造函数调用语句是静态方法、静态初始化语句、静态变量的变量初始化语句或显式构造函数调用语句（8.8.7 节）时，语句或表达式才会出现在静态上下文中。

如果类 `O` 是内部类 `C` 的直接词汇封闭类，并且 `C` 的声明没有出现在静态上下文中，则类 `C` 是类 `O` 的直接内部类。如果类 `C` 是类 `O` 的直接内部类，或者是类 `O` 的一个内部类的内



部类，则类  $C$  是类  $O$  的内部类。

类  $O$  是其自身的第 0 个词汇封闭类。如果类  $O$  是类  $C$  的第  $n-1$  个词汇封闭类的直接封闭类，则类  $O$  是类  $C$  的第  $n$  个词汇封闭类。

类  $O$  的直接内部类  $C$  的实例  $i$  与  $O$  的一个实例相关联，该实例称为  $i$  的直接封闭实例。对象的直接封闭实例（如果有的话）是在创建对象时确定的（15.9.2 节）。

对象  $o$  是其自身的第 0 个词汇封闭实例。如果对象  $o$  是实例  $i$  的第  $n-1$  个词汇封闭实例的直接封闭实例，则对象  $o$  是实例  $i$  的第  $n$  个词汇封闭实例。

当一个内部类引用一个实例变量，并且该变量是一个词汇封闭类的成员时，就会使用相应的词汇封闭实例的变量。不能在内部类中对词汇封闭类的 `blank final`（4.12.4 节）字段赋值。

其声明出现在静态上下文中的内部类  $I$  的实例不具有词汇封闭实例。但是，如果  $I$  是在静态方法或静态初始化语句内直接声明的，那么  $I$  确实具有一个封闭块语句，它是从词汇上封闭  $I$  的声明的最内层块语句。

此外，对于  $C$ （ $C$  自身是类  $SO$  的直接内部类）的每一个超类  $S$ ，都有一个与  $i$  关联的  $SO$  的实例，称为  $i$  关于  $S$  的直接封闭实例。关于其类的直接超类的对象的直接封闭实例（如果有的话）是在通过显式构造函数调用语句调用超类构造函数时确定的。

在内部类中使用但未声明的任何局部变量、形式方法参数或异常处理程序参数都必须声明为 `final`。在内部类中使用但未声明的任何局部变量都必须在内部类的主体之前明确进行赋值。

内部类包括本地（14.3 节）、匿名（15.9.5 节）和非静态成员类（8.5 节）。下面是一些示例：

```
class Outer {
    int i = 100;
    static void classMethod() {
        final int l = 200;
        class LocalInStaticContext {
            int k = i; // compile-time error
            int m = l; // ok
        }
    }
    void foo() {
        class Local { // a local class
            int j = i;
        }
    }
}
```

类 `LocalInStaticContext` 的声明出现在静态上下文中——在静态方法 `classMethod` 内部。类 `Outer` 的实例变量在静态方法的主体内不可用。特别地，`Outer` 的实例变量在 `LocalInStaticContext` 的主体内部不可用。但是，周围方法中的局部变量可以无错地被引用（假定它们被标记为 `final`）。

其声明未出现在静态上下文中的内部类可以自由地引用它们的封闭类的实例变量。实例变量总是关于一个实例定义的。就封闭类的实例变量而言，必须关于那个类的一个封闭实例来定义实例变量。因此，例如，上述类 `Local` 具有类 `Outer` 的一个封闭实例。一个更进一步的示例如下：

```
class WithDeepNesting{
    boolean toBe;
    WithDeepNesting(boolean b) { toBe = b;}
    class Nested {
        boolean theQuestion;
        class DeeplyNested {
            DeeplyNested(){
                theQuestion = toBe || !toBe;
            }
        }
    }
}
```

其中，`WithDeepNesting.Nested.DeeplyNested` 的每个实例都有类 `WithDeepNesting.Nested` 的一个封闭实例（它的直接封闭实例）和类 `WithDeepNesting` 的一个封闭实例（它的第 2 个词汇封闭实例）。

### 8.1.4 超类和子类

普通类声明中可选的 `extends` 子句指定了当前类的直接超类。

*Super:*

`extends ClassType`

下面重复介绍了第 4.3 节的内容，以使这里的介绍更清楚：

*ClassType:*

`TypeDeclSpecifier TypeArgumentsopt`

一个类被称为它的直接超类的直接子类。当前的类实现是从其直接超类的实现派生而来的。枚举类型 `E` 的直接超类是 `Enum<E>`。`extends` 子句绝对禁止出现在类 `Object` 的定义中，因为 `Object` 类是根类，没有任何直接超类。

给定  $C\langle F_1, \dots, F_n \rangle$ （其中  $n \geq 0$ ,  $C \neq \text{Object}$ ）的（可能是泛型）类声明，类类型（4.5 节） $C\langle F_1, \dots, F_n \rangle$  的直接超类是  $C$  的声明的 `extends` 子句（如果 `extends` 子句存在的话）中给定的类型；否则，它就是 `Object`。

设  $C\langle F_1, \dots, F_n \rangle$ （其中  $n > 0$ ）是泛型类声明。参数化类类型  $C\langle T_1, \dots, T_n \rangle$ （其中  $T_i$ ,  $1 \leq i \leq n$ ，是一个类型，的直接超类是  $D\langle U_1 \text{ theta}, \dots, U_k \text{ theta} \rangle$ （其中  $D\langle U_1, \dots, U_k \rangle$  是  $C\langle F_1, \dots, F_n \rangle$  的直接超类）， $\text{theta}$  是代换  $[F_1 := T_1, \dots, F_n := T_n]$ 。

*ClassType* 必须指定一个可访问的（6.6 节）类类型，否则就会发生编译时错误。如果指定的 *ClassType* 指定了一个 `final` 类（8.1.1.2 节），则会发生编译时错误；`final` 类不

允许具有子类。如果 *ClassType* 指定了类 *Enum* 或者它的任何调用，则会发生编译时错误。如果 *TypeName* 后接任何类型参数，则它必须是 *TypeName* 指定的类型声明的正确调用，并且任何类型参数都不能是通配符类型参数，否则将会发生编译时错误。

在下面的示例中：

```
class Point { int x, y; }
final class ColoredPoint extends Point { int color; }
class Colored3dPoint extends ColoredPoint { int z; } // error
```

其关系如下：

- 类 *Point* 是 *Object* 的直接子类。
- 类 *Object* 是类 *Point* 的直接超类。
- 类 *ColoredPoint* 是类 *Point* 的直接子类。
- 类 *Point* 是类 *ColoredPoint* 的直接超类。

类 *Colored3dPoint* 的声明会引发编译时错误，这是由于它试图扩展 *final* 类 *ColoredPoint*。

子类关系是直接子类关系的传递闭包 (transitive closure)。如果以下条件之一成立，则类 *A* 是类 *C* 的子类：

- *A* 是 *C* 的直接子类。
- 存在一个类 *B*，其中类 *A* 是类 *B* 的子类，并且 *B* 是 *C* 的子类，递归地应用这个定义。无论何时类 *A* 是类 *C* 的子类，则称类 *C* 是类 *A* 的超类。

在下面的示例中：

```
class Point { int x, y; }
class ColoredPoint extends Point { int color; }
final class Colored3dPoint extends ColoredPoint { int z; }
```

其关系如下：

- 类 *Point* 是类 *ColoredPoint* 的超类。
- 类 *Point* 是类 *Colored3dPoint* 的超类。
- 类 *ColoredPoint* 是类 *Point* 的子类。
- 类 *ColoredPoint* 是类 *Colored3dPoint* 的超类。
- 类 *Colored3dPoint* 是类 *ColoredPoint* 的子类。
- 类 *Colored3dPoint* 是类 *Point* 的子类。

如果类型 *T* 在类 *C* 的 *extends* 或 *implements* 子句中被称为超类或超接口，或者被称为超类或超接口名称的限定符，则类 *C* 直接依赖于类型 *T*。如果以下条件之一成立，则类 *C* 依赖于引用类型 *T*：

- *C* 直接依赖于 *T*。
- *C* 直接依赖于接口 *I*，而 *I* 依赖 (9.1.3 节) 于 *T*。
- *C* 直接依赖于类 *D*，而 *D* 依赖于 *T* (递归地使用这个定义)。

如果一个类依赖于它自身，则会发生编译时错误。

例如：

```
class Point extends ColoredPoint { int x, y; }
class Co
loredPoint extends Point { int color; }
```

会引发编译时错误。

如果在运行时检测到循环声明的类，则在加载类（12.2 节）时会抛出一个 `ClassCircularityError`。

### 8.1.5 超接口

类声明中可选的 `implements` 子句列出了接口的名称，该接口是正被声明的类的直接超接口：

*Interfaces:*

`implements InterfaceTypeList`

*InterfaceTypeList:*

*InterfaceType*

*InterfaceTypeList*, *InterfaceType*

下面重复介绍了第 4.3 节的内容，以使这里的介绍更清楚：

*InterfaceType:*

*TypeDeclSpecifier TypeArguments<sub>opt</sub>*

给定  $C\langle F_1, \dots, F_n \rangle$ （其中  $n \geq 0$ ,  $C \neq \text{Object}$ ）的（可能是泛型）类声明，类类型（4.5 节） $C\langle F_1, \dots, F_n \rangle$  的直接超接口是  $C$  的声明的 `implements` 子句（如果 `implements` 子句存在的话）中给定的类型。

设  $C\langle F_1, \dots, F_n \rangle$ （其中  $n > 0$ ）是泛型类声明。参数化类类型  $C\langle T_1, \dots, T_n \rangle$ （其中  $T_i, 1 \leq i \leq n$ , 是一个类型）的直接超接口是  $I\langle U_1 \text{ theta}, \dots, U_k \text{ theta} \rangle$ （其中  $I\langle U_1, \dots, U_k \rangle$  是  $C\langle F_1, \dots, F_n \rangle$  的直接超接口）， $\text{theta}$  是代换  $[F_1 := T_1, \dots, F_n := T_n]$ 。

每个 *InterfaceType* 都必须指定一个可访问的（6.6 节）接口类型，否则就会发生编译时错误。如果 *TypeName* 后接任何类型参数，则它必须是 *TypeName* 指定的类型声明的正确调用，并且任何类型参数都不会是通配符类型参数，否则就会发生编译时错误。

如果在单一 `implements` 子句名称中，两次或多次把相同的接口称为直接超接口，则会发生编译时错误。

事实确实如此，即使以不同的方式命名接口；例如，下面的代码：

```
class Redundant implements java.lang.Cloneable, Cloneable {
    int x;
}
```

会导致一个编译时错误，这是由于名称 `java.lang.Cloneable` 和 `Cloneable` 引用相同的接口。如果以下条件之一成立，则接口类型  $I$  是类类型  $C$  的超接口：

- $I$  是  $C$  的直接超接口。
- $C$  具有某个直接超接口  $J$ ， $I$  则是  $J$  的超接口，使用第 9.1.3 节中给出的“接口的超

接口”的定义。

- *I* 是 *C* 的直接超类的超接口。

类被称为实现其所有的超接口。

在下面的示例中：

```
public interface Colorable {
    void setColor(int color);
    int getColor();
}

public enum Finish {MATTE, GLOSSY}

public interface Paintable extends Colorable {
    void setFinish(Finish finish);
    Finish getFinish();
}

class Point { int x, y; }

class ColoredPoint extends Point implements Colorable {
    int color;
    public void setColor(int color) { this.color = color; }
    public int getColor() { return color; }
}

class PaintedPoint extends ColoredPoint implements Paintable {
    Finish finish;
    public void setFinish(Finish finish) {
        this.finish = finish;
    }
    public Finish getFinish() { return finish; }
}
```

其关系如下：

- 接口 Paintable 是类 PaintedPoint 的超接口。
- 接口 Colorable 是类 ColoredPoint 和类 PaintedPoint 的超接口。
- 接口 Paintable 是接口 Colorable 的子接口，且 Colorable 是 Paintable 的超接口，如第 9.1.3 节中的定义。

类可以用不止一种方式拥有一个超接口。在本示例中，类 PaintedPoint 具有超接口 Colorable，这是由于 Colorable 是 ColoredPoint 的超接口，并且由于 Colorable 还是 Paintable 的超接口。除非被声明的类是 abstract，否则每个直接超接口的所有方法成员的声明都必须被该类中的声明实现，或者通过继承自直接超类的现有方法声明来实现，因为不是 abstract 的类不允许具有 abstract 方法（8.1.1.1 节）。

因此，以下示例：

```
interface Colorable {
    void setColor(int color);
    int getColor();
}

class Point { int x, y; };

class ColoredPoint extends Point implements Colorable {
```



```
    int color;
}
```

会引发一个编译时错误，这是因为 `ColoredPoint` 不是一个 `abstract` 类，但是它不能提供接口 `Colorable` 的方法 `setColor` 和 `getColor` 的实现。

允许类中的单一方法声明实现多个超接口的方法。例如，在下面的代码中：

```
interface Fish { int getNumberOfScales(); }
interface Piano { int getNumberOfScales(); }
class Tuna implements Fish, Piano {
    // You can tune a piano, but can you tuna fish?
    int getNumberOfScales() { return 91; }
}
```

类 `Tuna` 中的方法 `getNumberOfScales` 的名称、签名和返回类型与接口 `Fish` 中声明的方法匹配，还与接口 `Piano` 中声明的方法匹配；可认为它实现了这两个方法。

另一方面，在如下这种情形中：

```
interface Fish { int getNumberOfScales(); }
interface StringBass { double getNumberOfScales(); }
class Bass implements Fish, StringBass {
    // This declaration cannot be correct, no matter what type is used.
    public ??? getNumberOfScales() { return 91; }
}
```

不可能声明一个名为 `getNumberOfScales` 并且其签名和返回类型与接口 `Fish` 和 `StringBass` 中声明的两个方法的签名和返回类型兼容的方法，这是因为一个类不能具有多个有相同签名和不同基本返回类型的方法（8.4 节）。因此，单独一个类不可能同时实现接口 `Fish` 和 `StringBass`（8.4.8 节）。

如果两个接口类型是相同泛型接口（9.1.2 节）的不同调用，或者是泛型接口的调用和指定相同泛型接口的原生类型，那么一个类不能同时是这两个接口类型的子类型。

下面是接口的非法多重继承的示例：

```
class B implements I<Integer>
class C extends B implements I<String>
```

引入这种要求是为了支持通过类型擦除（4.6 节）进行转换。

### 8.1.6 类体和成员声明

类体可以包含类的成员，即字段（8.3 节）、类（8.5 节）、接口（8.5 节）和方法（8.4 节）的声明。类体还可以包含类的实例初始化语句（8.6 节）、静态初始化语句（8.7 节）和构造函数（8.8 节）的声明。

*ClassBody:*

```
{ ClassBodyDeclarationsopt }
```

```

ClassBodyDeclarations:
    ClassBodyDeclaration
    ClassBodyDeclarations ClassBodyDeclaration
ClassBodyDeclaration:
    ClassMemberDeclaration
    InstanceInitializer
    StaticInitializer
    ConstructorDeclaration
ClassMemberDeclaration:
    FieldDeclaration
    MethodDeclaration
    ClassDeclaration
    InterfaceDeclaration
;

```

在类类型 *C* 中声明或被其继承的成员 *m* 声明的作用域是 *C* 的整个主体，包括任何嵌套的类型声明。

如果 *C* 自身是一个嵌套类，则在封闭作用域中可能有与 *m* 相同种类（变量、方法或类型）的定义和名称（这些作用域可能是块语句、类或包）。在所有这些情况下，在 *C* 中声明或继承的成员 *m* 会屏蔽（6.3.1 节）相同类型和名称的其他定义。

## 8.2 类成员

我不想加入任何愿意接纳我为其会员的俱乐部。  
——Groucho Marx（电影人）

类类型的所有成员如下：

- 继承自其直接超类（8.1.4 节）的成员，类 `Object` 除外，它没有直接超类
- 继承自任何直接超接口（8.1.5 节）的成员
- 在类体（8.1.6 节）中声明的成员

被声明为 `private` 的类成员不能被该类的子类继承。除了声明类的包之外，在其他包中声明的子类只可以继承那些被声明为 `protected` 或 `public` 的类成员。

我们使用短语“成员的类型”来指示：

- 其类型（对于字段）。
- 一个有序的三元组（对于方法），包括：
  - ◆ **参数类型**：方法成员的参数类型的列表。
  - ◆ **返回类型**：方法成员的返回类型。
  - ◆ **throws 子句**：方法成员的 `throws` 子句中声明的异常类型。

构造函数、静态初始化语句和实例初始化语句都不是成员，因此都不能被继承。下面的示例：

```
class Point {
    int x, y;
    private Point() { reset(); }
    Point(int x, int y) { this.x = x; this.y = y; }
    private void reset() { this.x = 0; this.y = 0; }
}
class ColoredPoint extends Point {
    int color;
    void clear() { reset(); } // error
}
class Test {
    public static void main(String[] args) {
        ColoredPoint c = new ColoredPoint(0, 0); // error
        c.reset(); // error
    }
}
```

将引发 4 个编译时错误：

- 由于 ColoredPoint 没有用两个整型参数声明的构造函数，就像 main 中的使用所要求的那样，因此会发生一个错误。这说明了 ColoredPoint 没有继承其超类 Point 的构造函数这一事实。
- 由于 ColoredPoint 没有声明任何构造函数，因此会为其自动创建一个默认构造函数（8.8.9 节），并且该默认构造函数等价于：

```
ColoredPoint() { super(); }
```

它会在不使用参数的情况下调用类 ColoredPoint 的直接超类的构造函数，因而会发生另一个错误。该错误的原因在于 Point 的构造函数不会获取 private 类型参数，因此在 Point 外部不可访问，甚至通过超类构造函数调用也是如此（8.8.7 节）。

还会发生另外两个错误，这是由于类 Point 的方法 reset 是 private 类型，因此不会被类 ColoredPoint 继承。因此，类 ColoredPoint 的方法 clear 和类 Test 的方法 main 中的方法调用不正确。

## 8.2.1 继承的示例

本节通过几个示例阐释了类成员的继承。

### 8.2.1.1 示例：通过默认访问继承

考虑一个示例，其中 points 包声明了下面两个编译单元：

```
package points;
public class Point {
    int x, y;
    public void move(int dx, int dy) { x += dx; y += dy; }
}
```

和:

```
package points;
public class Point3d extends Point {
    int z;
    public void move(int dx, int dy, int dz) {
        x += dx; y += dy; z += dz;
    }
}
```

以及另一个包中的第三个编译单元, 如下:

```
import points.Point3d;
class Point4d extends Point3d {
    int w;
    public void move(int dx, int dy, int dz, int dw) {
        x +=dx;y +=dy; z +=dz; w +=dw; // compile-time errors
    }
}
```

这里, points 包中的两个类会通过编译。类 Point3d 继承类 Point 的字段 x 和 y, 因为它与 Point 位于相同的包中。类 Point4d (它位于不同的包中) 不会继承类 Point 的字段 x 和 y, 或者类 Point3d 的字段 z, 因此不会通过编译。

编写第三个编译单元的一种更好的方式如下:

```
import points.Point3d;
class Point4d extends Point3d {
    int w;
    public void move(int dx, int dy, int dz, int dw) {
        super.move(dx, dy, dz); w += dw;
    }
}
```

它使用超类 Point3d 的 move 方法处理 dx、dy 和 dz。如果用这种方式编写 Point4d, 则会无错地通过编译。

### 8.2.1.2 通过 public 和 protected 继承

给出类 Point:

```
package points;
public class Point {
    public int x, y;
    protected int useCount = 0;
    static protected int totalUseCount = 0;
    public void move(int dx, int dy) {
        x += dx; y += dy; useCount++; totalUseCount++;
    }
}
```

在 Point 的子类中会继承 public 和 protected 字段 x、y、useCount 和 totalUse Count。

因此, 本测试程序 (在另一个包中) 会成功地通过编译:

```
class Test extends points.Point {
    public void moveBack(int dx, int dy) {
```

```
        x -= dx; y -= dy; useCount++; totalUseCount++;
    }
}
```

### 8.2.1.3 通过 private 继承

在下面的示例中：

```
class Point {
    int x, y;
    void move(int dx, int dy) {
        x += dx; y += dy; totalMoves++;
    }
    private static int totalMoves;
    void printMoves() { System.out.println(totalMoves); }
}
class Point3d extends Point {
    int z;
    void move(int dx, int dy, int dz) {
        super.move(dx, dy); z += dz; totalMoves++;
    }
}
```

类变量 `totalMoves` 只能在类 `Point` 内部使用；它不会被子类 `Point3d` 继承。由于类 `Point3d` 的方法 `move` 尝试递增 `totalMoves`，因此会发生编译时错误。

### 8.2.1.4 访问不可访问类的成员

即使一个类可能未被声明为 `public`，通过 `public` 超类或超接口声明该类的包外部的代码也有可能在运行时使用该类的实例。可以把该类的实例赋予 `public` 类型这样的变量。如果该类实现或重写了 `public` 超类或超接口的方法，则通过这种变量引用的对象的 `public` 方法的调用可以调用该类的方法（在这种情形下，该方法必须被声明为 `public`，即使它是在非 `public` 类中声明的也是如此）。

考虑下面的编译单元：

```
package points;
public class Point {
    public int x, y;
    public void move(int dx, int dy) {
        x += dx; y += dy;
    }
}
```

另一个包的另一个编译单元如下：

```
package morePoints;
class Point3d extends points.Point {
    public int z;
    public void move(int dx, int dy, int dz) {
        super.move(dx, dy); z += dz;
    }
    public void move(int dx, int dy) {
```



```

        move(dx, dy, 0);
    }
}
public class OnePoint {
    public static points.Point getOne() {
        return new Point3d();
    }
}

```

仍然在第三个包中的一个调用 `morePoints.OnePoint.getOne()` 将返回一个 `Point3d`，它可以用作一个 `Point`，即使类型 `Point3d` 在包 `morePoints` 外部不可用。这样，可以为那个对象调用方法 `move` 的两个参数的版本，这是允许的，因为 `Point3d` 的方法 `move` 是 `public` 类型（因为它必须是这种类型，这是由于重写 `public` 方法的任何方法自身都必须是 `public` 类型，正是如此，像这样的情形将会正确地工作）。还可以从像第三个包那样的包中访问那个对象的字段 `x` 和 `y`。

虽然类 `Point3d` 的字段 `z` 是 `public` 类型，但是不可能通过包 `morePoints` 外部的代码访问该字段，假定只有一个引用指向类型 `Point` 的变量 `p` 中的类 `Point3d` 的一个实例。这是由于表达式 `p.z` 是不正确的，因为 `p` 具有类型 `Point`，并且类 `Point` 没有名为 `z` 的字段；此外，表达式 `((Point3d)p).z` 也不正确，因为类类型 `Point3d` 不能在包 `morePoints` 外部被引用。

但是，把字段 `z` 声明为 `public` 并不是没有用处的。如果在包 `morePoints` 中有类 `Point3d` 的一个 `public` 子类 `Point4d`：

```

package morePoints;
public class Point4d extends Point3d {
    public int w;
    public void move(int dx, int dy, int dz, int dw) {
        super.move(dx, dy, dz); w += dw;
    }
}

```

那么类 `Point4d` 将继承字段 `z`，该字段是 `public` 类型，因此可以被除了 `morePoints` 之外的其他包中的代码通过 `public` 类型 `Point4d` 的变量和表达式进行访问。

## 8.3 字段声明

诗情画意包围着我，我看上去仍然徜徉在文艺胜地。  
——Joseph Addison (1672~1719), 《A Letter from Italy》

类类型的变量是通过字段声明引入的：

*FieldDeclaration:*

*FieldModifiers*<sub>opt</sub> *Type* *VariableDeclarators* ;

*VariableDeclarators:*

*VariableDeclarator*

*VariableDeclarators* , *VariableDeclarator*

*VariableDeclarator:*

*VariableDeclaratorId*

*VariableDeclaratorId* = *VariableInitializer*

*VariableDeclaratorId:*

*Identifier*

*VariableDeclaratorId* [ ]

*VariableInitializer:*

*Expression*

*ArrayInitializer*

第 8.3.1 节中描述了 *FieldModifiers*。可以在名称中使用 *FieldDeclarator* 中的 *Identifier* 来引用字段。字段是成员；第 8.1.6 节中指定了字段声明的作用域（6.3 节）。通过使用多个声明符可以在单一字段声明中声明多个字段；*FieldModifiers* 和 *Type* 适用于声明中的所有声明符。第 10.2 节中讨论了涉及数组类型的变量声明。

如果类声明的主体中声明了两个同名字段，则会发生编译时错误。方法、类型和字段可以具有相同的名称，因为它们使用在不同的上下文中，并且通过不同的查找过程消除歧义（6.5 节）。

如果类用某个名称声明了一个字段，就称该字段的声明隐藏了那个类的超类和超接口中具有相同名称字段的任何和所有可访问的声明。字段声明还会屏蔽（6.3.1 节）封闭类或接口中的任何可访问的字段的声明，以及任何封闭块语句中具有相同名称的任何局部变量、形式方法参数和异常处理程序参数的声明。

如果字段声明隐藏了另一个字段的声明，那么这两个字段不必具有相同的类型。

一个类可以继承其直接超类和直接超接口，超类和超接口中的所有非私有字段对于该类的代码都是可访问的，并且不会被类中的声明隐藏。

注意，超类的私有字段可以被子类访问（例如，如果两个类都是同一个类的成员）。然而，私有字段永远不会被子类继承。

类有可能继承多个具有相同名称的字段（8.3.3.3 节）。这样的情形本质上不会引发编译时错误。但是，在类体内试图通过其简单名称引用任何这样的字段，都会导致编译时错误，这是因为这样的引用是有歧义的。

可以通过多条途径从一个接口继承相同的字段声明。在这种情形下，将字段视为只会被继承一次，并且可以通过其简单名称无歧义地引用它。

可以使用限定名称（如果字段是 *static*）或者使用字段访问表达式（它包含关键字 *super* 或转换到超类类型的强制转换）（15.11 节）来访问隐藏的字段。有关讨论和示例，参见第 15.11.2 节。

存储在 *float* 类型的字段中的值总是浮点值集（4.2.3 节）的元素；类似地，存储在

`double` 类型的字段中的值总是双精度值集的元素。`float` 类型的字段不允许包含并非浮点值集元素的浮点扩展的指数值集元素，`double` 类型的字段也不允许包含并非双精度值集元素的双精度扩展的指数值集元素。

### 8.3.1 字段修饰符

*FieldModifiers:*

*FieldModifier*

*FieldModifiers FieldModifier*

*FieldModifier: one of*

`Annotation public protected private`  
`static final transient volatile`

第 6.6 节中讨论了访问修饰符 `public`、`protected` 和 `private`。如果相同的修饰符在一个字段声明中出现了不止一次，或者如果一个字段声明中具有多个访问修饰符 `public`、`protected` 和 `private`，则会发生编译时错误。

如果字段声明上的注释 `a` 对应于一个注释类型 `T`，并且 `T` 具有（元）注释 `m`，它对应于 `annotation.Target`，那么 `m` 必须具有一个其值为 `annotation.ElementType.FIELD` 的元素，否则就会发生编译时错误。第 9.7 节中进一步描述了注释修饰符。

如果字段声明中出现了两个或多个（不同的）字段修饰符，那么习惯上让它们出现的顺序与上面显示的 *FieldModifier* 的产生式中的顺序一致，尽管不需要如此。

#### 8.3.1.1 static 字段

如果字段被声明为 `static`，则刚好只有一个该字段的具体化，而不管最终创建了类的多少个实例（可能是零个）。`static` 字段（有时称为类变量）是在类被初始化（12.4 节）时具体化的。

未被声明为 `static` 的字段（有时称为非 `static` 字段）被称为实例变量。无论何时创建类的一个新实例，都会为该类或其任何超类中声明的每个实例变量创建一个与该实例关联的新变量。示例程序：

```
class Point {
    int x, y, useCount;
    Point(int x, int y) { this.x = x; this.y = y; }
    final static Point origin = new Point(0, 0);
}
class Test {
    public static void main(String[] args) {
        Point p = new Point(1,1);
        Point q = new Point(2,2);
        p.x = 3; p.y = 3; p.useCount++; p.origin.useCount++;
        System.out.println("(" + q.x + ", " + q.y + ")");
        System.out.println(q.useCount);
    }
}
```

```
        System.out.println(q.origin == Point.origin);  
        System.out.println(q.origin.useCount);  
    }  
}
```

输出结果如下:

```
(2,2)  
0  
true  
1
```

这表明更改 `p` 的字段 `x`、`y` 和 `useCount` 不会影响 `q` 的字段, 因为这些字段是不同对象中的实例变量。在这个示例中, 同时使用类名作为限定符 (如 `Point.origin`) 和字段访问表达式 (15.11 节) 中的类类型的变量 (如 `p.origin` 和 `q.origin`) 来引用类 `Point` 的类变量 `origin`。这两种访问 `origin` 类变量的方式访问的是相同的对象, 这可以通过引用相等性表达式 (15.21.3 节) 的值这一事实来证明:

```
q.origin==Point.origin
```

为真。进一步的证据是下面的递增表达式:

```
p.origin.useCount++;
```

导致 `q.origin.useCount` 的值为 1; 这是由于 `p.origin` 和 `q.origin` 引用相同的变量。

#### 8.3.1.2 final 字段

字段可以声明为 `final` (4.12.4 节)。类变量 (`static` 字段) 和实例变量 (非 `static` 字段) 都可以声明为 `final`。

如果在类中声明一个 `blank final` (4.12.4 节) 类变量时, 未在该类中通过静态初始化语句 (8.7 节) 对该变量明确赋值 (16.8 节), 则会发生编译时错误。

在声明 `blank final` 实例变量的类中, 必须在该类的每个构造函数 (8.8 节) 末尾明确对该变量赋值 (16.9 节); 否则, 就会发生编译时错误。

#### 8.3.1.3 transient 字段

变量可以被标记为 `transient`, 以表明它们不是对象的持久状态的一部分。

如果类 `Point` 的一个实例:

```
class Point {  
    int x, y;  
    transient float rho, theta;  
}
```

通过系统服务保存到永久性存储器, 那么只会保存字段 `x` 和 `y`。本规范没有详细说明这种服务的细节; 有关这种服务的示例, 请参见 `java.io.Serializable` 的规范。

#### 8.3.1.4 volatile 字段

如第 17 章中所述, Java 编程语言允许线程访问共享变量。一条规则是: 为了确保共享变量会被一致地和可靠地更新, 线程应该通过获得一个锁为那些共享变量强制执行互斥,

来确保它独占地使用这种变量。

Java 编程语言提供了第二种机制, 即 `volatile` 字段, 它较出于某些目的而执行锁定更方便。

字段可以声明为 `volatile`, 在这种情况下, Java 内存模型 (第 17 章) 将确保所有的线程会看到该变量的一致值。

在下面的示例中, 如果一个线程反复调用方法 `one` (但是总共不超过 `Integer.MAX_VALUE` 次), 另一个线程反复调用方法 `two`:

```
class Test {
    static int i = 0, j = 0;
    static void one() { i++; j++; }
    static void two() {
        System.out.println("i=" + i + " j=" + j);
    }
}
```

那么方法 `two` 可能偶尔输出一个比 `i` 值大的 `j` 值, 这是由于本示例不包括同步, 并且在第 17 章解释的规则之下, 可能无序地更新 `i` 和 `j` 的共享值。

防止这种无序行为的一种方式是把方法 `one` 和 `two` 声明为 `synchronized` (8.4.3.6 节):

```
class Test {
    static int i = 0, j = 0;
    static synchronized void one() { i++; j++; }
    static synchronized void two() {
        System.out.println("i=" + i + " j=" + j);
    }
}
```

这将阻止方法 `one` 和方法 `two` 并发执行, 并且进一步保证在方法 `one` 返回之前 `i` 和 `j` 的共享值都会得到更新。因此, 方法 `two` 永远不会观察到一个大于 `i` 的 `j` 值。

另一种方法是把 `i` 和 `j` 声明为 `volatile`:

```
class Test {
    static volatile int i = 0, j = 0;
    static void one() { i++; j++; }
    static void two() {
        System.out.println("i=" + i + " j=" + j);
    }
}
```

这将允许方法 `one` 和方法 `two` 并发执行, 但是会保证在每个线程执行程序代码期间, 对 `i` 和 `j` 的共享值的访问将会刚好出现那么多次, 并且访问的顺序刚好与它们出现的顺序相同。因此, `j` 的共享值永远不会大于 `i` 的共享值, 这是由于在更新 `j` 之前, 对 `i` 的每次更新都必须反映在 `i` 的共享值中。但是, 方法 `two` 的任何给定的调用有可能观察到 `j` 值比观察到的 `i` 值大得多, 这是由于在方法 `two` 获取 `i` 值的那一刻与方法 `two` 获取 `j` 值的那一刻之间, 方法 `one` 可能执行许多次。

有关更多的讨论和示例, 参见第 17 章。

如果 `final` 变量还被声明为 `volatile`, 则会发生编译时错误。

### 8.3.2 字段的初始化

如果字段声明符包含一个变量初始化语句，那么它就具有给声明的变量赋值的语义（15.26 节），并且：

- 如果声明符用于类变量（即 `static` 字段），那么在初始化类（12.4 节）时，就会对变量初始化语句进行求值，并正好执行一次赋值。
- 如果声明符用于实例变量（即非 `static` 字段），那么每当创建类的一个实例（12.5 节）时，就会对变量初始化语句求值，并执行赋值。

下面的示例：

```
class Point {  
    int x = 1, y = 5;  
}  
  
class Test {  
    public static void main(String[] args) {  
        Point p = new Point();  
        System.out.println(p.x + ", " + p.y);  
    }  
}
```

产生如下输出：

```
1, 5
```

这是由于无论何时创建一个新的 `Point`，都会对 `x` 和 `y` 进行赋值。

变量初始化语句还用在局部变量声明语句（14.4 节）中，每当执行局部变量声明语句时，都会对初始化语句进行求值并执行赋值。

如果命名类（或接口）的 `static` 字段的变量初始化语句的求值可能突然通过受查的异常（11.2 节）完成，则会发生编译时错误。

如果命名类的实例变量初始化语句可能抛出一个受查异常，那么除非该异常或其子类型之一是在其类的每个构造函数的 `throws` 子句中显式声明的，并且该类至少具有一个显式声明的构造函数，否则会发生编译时错误。匿名类（15.9.5 节）中的实例变量初始化语句可以抛出任何异常。

#### 8.3.2.1 类变量的初始化语句

如果在类变量的初始化表达式中，通过简单名称引用任何实例变量，则会发生编译时错误。

如果类变量的初始化表达式中出现了关键字 `this`（15.8.3 节）或关键字 `super`（15.11.2 节、15.12 节），则会发生编译时错误。

这里的一个微妙之处是：在运行时，具有 `final` 类型并通过编译时常量值进行初始化的 `static` 变量会最先初始化。这也适用于接口中的字段（9.3.1 节）。这些变量都是永远不会被观察到具有其默认初始值（4.12.5 节）的“常量”，甚至通过迂回的程序也是如此。有关更多讨论，参见第 12.4.2 节和第 13.4.9 节。



有时会限制在声明之前使用类变量，即使这些类变量都在作用域中。有关管理指向类变量的向前引用的确切规则，参见第 8.3.2.3 节。

#### 8.3.2.2 实例变量的初始化语句

实例变量的初始化表达式可以使用类中声明的或被该类继承的任何 static 变量的简单名称，即使变量的声明出现在代码后面也是如此。

因此，下面的示例：

```
class Test {  
    float f = j;  
    static int j = 1;  
}
```

会无错地通过编译，当初始化类 Test 时，它会将 j 初始化为 1，并在每次创建类 Test 的实例时，将 f 初始化为 j 的当前值。

实例变量的初始化表达式允许引用当前对象 this(15.8.3 节)，以及使用关键字 super(15.11.2 节、15.12 节)。

有时会限制在声明之前使用实例变量，即使这些实例变量都在作用域中。有关管理指向实例变量的向前引用的确切规则，参见 8.3.2.3 节。

#### 8.3.2.3 初始化期间有关使用字段的限制

仅当成员是类或接口 C 的实例（或 static）字段，并且下列所有条件均成立时，成员的声明才需要出现在使用它的代码之前：

- 使用发生在 C 的实例（或 static）变量初始化语句中，或者发生在 C 的实例（或 static）初始化语句中。
- 使用不在赋值的左边。
- 通过简单名称使用。
- C 是封闭使用的最内层的类或接口。

如果上述 4 个要求中有任何一个不满足，则会发生编译时错误。

这意味着以下测试程序将会产生编译时错误：

```
class Test {  
    int i = j; // compile-time error: incorrect forward reference  
    int j = 1;  
}
```

而下列示例将会无错地通过编译：

```
class Test {  
    Test() { k = 2; }  
    int j = 1;  
    int i = j;  
    int k;  
}
```

即使 Test 的构造函数（8.8 节）引用的字段 k 是在三行代码之后声明的也是如此。

这些限制被设计成在编译时捕获循环或异常的初始化。因此，下列示例程序：

```
class Z {
    static int i = j + 2;
    static int j = 4;
}
```

和:

```
class Z {
    static { i = j + 2; }
    static int i, j;
    static { j = 4; }
}
```

将导致编译时错误。这样，方法执行的访问都不会得到检查，因此：

```
class Z {
    static int peek() { return j; }
    static int i = peek();
    static int j = 1;
}
class Test {
    public static void main(String[] args) {
        System.out.println(Z.i);
    }
}
```

将产生如下输出：

0

因为在通过其变量初始化语句初始化 *j* 之前，*i* 的变量初始化语句会使用类方法 *peek* 访问变量 *j* 的值，此时，*j* 仍然具有其默认值（4.12.5 节）。

一个更加精心设计的示例如下：

```
class UseBeforeDeclaration {
    static {
        x = 100; // ok - assignment
        int y = x + 1; // error - read before declaration
        int v = x = 3; // ok - x at left hand side of assignment
        int z = UseBeforeDeclaration.x * 2;
        // ok - not accessed via simple name
        Object o = new Object() {
            void foo() { x++; } // ok - occurs in a different class
            { x++; } // ok - occurs in a different class
        };
    }
    {
        j = 200; // ok - assignment
        j = j + 1; // error - right hand side reads before declaration
        int k = j = j + 1;
        int n = j = 300; // ok - j at left hand side of assignment
        int h = j++; // error - read before declaration
        int l = this.j * 3; // ok - not accessed via simple name
    }
}
```

```

Object o = new Object(){
    void foo(){j++;} // ok - occurs in a different class
    { j = j + 1;} // ok - occurs in a different class
};
}

int w = x = 3; // ok - x at left hand side of assignment
int p = x; // ok - instance initializers may access static fields
static int u = (new Object(){int bar(){return x;}}).bar();
// ok - occurs in a different class
static int x;
int m = j = 4; // ok - j at left hand side of assignment
int o = (new Object(){int bar(){return j;}}).bar();
// ok - occurs in a different class
int j;
}

```

### 8.3.3 字段声明的示例

下列示例阐释了关于字段声明的一些（可能是微妙的）要点。

#### 8.3.3.1 示例：类变量的隐藏

下面的示例：

```

class Point {
    static int x = 2;
}
class Test extends Point {
    static double x = 4.7;
    public static void main(String[] args) {
        new Test().printX();
    }
    void printX() {
        System.out.println(x + " * " + super.x);
    }
}

```

将产生如下输出：

```
4.7 2
```

这是由于类 Test 中 x 的声明隐藏了类 Point 中 x 的定义，因此类 Test 不会从其超类 Point 继承字段 x。在类 Test 的声明内，简单名称 x 引用类 Test 内声明的字段。类 Test 中的代码可以把类 Point 的字段作为 super.x 引用（或者，由于 x 是 static，因此作为 Point.x 引用）。如果删除 Test.x 的声明：

```

class Point {
    static int x = 2;
}
class Test extends Point {
    public static void main(String[] args) {

```

```

        new Test().printX();
    }
    void printX() {
        System.out.println(x + " * " + super.x);
    }
}

```

则类 Point 的字段 x 不再被隐藏在类 Test 之内；相反，简单名称 x 现在可以引用字段 Point.x。类 Test 中的代码仍然可以把相同的字段作为 super.x 引用。因此，该变体程序的输出如下：

```
2 2
```

### 8.3.3.2 示例：实例变量的隐藏

本示例类似于上一节中的那个示例，但是它使用的是实例变量，而不是静态变量。代码如下：

```

class Point {
    int x = 2;
}
class Test extends Point {
    double x = 4.7;
    void printBoth() {
        System.out.println(x + " * " + super.x);
    }
    public static void main(String[] args) {
        Test sample = new Test();
        sample.printBoth();
        System.out.println(sample.x + " * " +
                               ((Point)sample).x);
    }
}

```

产生如下输出：

```
4.7 2
```

```
4.7 2
```

这是由于类 Test 中 x 的声明隐藏了类 Point 中 x 的定义，因此类 Test 不会从其超类 Point 继承字段 x。但是，必须注意：虽然类 Point 的字段 x 未被类 Test 继承，然而它会被类 Test 的实例实现。换句话说，类 Test 的每个实例都包含两个字段：一个是 int 类型，一个是 double 类型。这两个字段都具有名称 x，但是在类 Test 的声明内，简单名称 x 总是引用类 Test 内声明的字段。类 Test 的实例方法中的代码可以把类 Point 的实例变量 x 作为 super.x 引用。

使用字段访问表达式访问字段 x 的代码，将访问通过引用表达式的类型指定的类中名为 x 的字段。因此，表达式 sample.x 访问的是一个 double 值，即类 Test 中声明的实例变量，这是由于变量 sample 的类型是 Test；但是表达式 ((Point)sample).x 访问的是一个 int 值，即类 Point 中声明的实例变量，这是由于转换到类型 Point 的强制转换。

如果从类 Test 中删除 x 的声明，如下列程序：

```

class Point {
    static int x = 2;
}
class Test extends Point {
    void printBoth() {
        System.out.println(x + " " + super.x);
    }
    public static void main(String[] args) {
        Test sample = new Test();
        sample.printBoth();
        System.out.println(sample.x + " " + ((Point)sample).x);
    }
}

```

则类 Point 的字段 x 不再被隐藏在类 Test 内。在类 Test 的声明中的实例方法内，简单名称 x 现在引用类 Point 内声明的字段。类 Test 中的代码仍然可以把那个相同的字段作为 super.x 引用。表达式 sample.x 仍然引用类型 Test 内的字段 x，但是该字段现在是一个继承的字段，因此会引用类 Point 中声明的字段 x。这个变体程序的输出如下：

```

2 2
2 2

```

### 8.3.3.3 示例：多重继承的字段

类可以从两个接口或者其超类以及一个接口继承两个或多个同名字段。如果试图通过其简单名称引用任何有歧义地继承的字段，则会发生编译时错误。可以使用限定名称或包含关键字 super (15.11.2 节) 的字段访问表达式无歧义地访问这样的字段。在下面的示例中：

```

interface Frob { float v = 2.0f; }
class SuperTest { int v = 3; }
class Test extends SuperTest implements Frob {
    public static void main(String[] args) {
        new Test().printV();
    }
    void printV() { System.out.println(v); }
}

```

类 Test 继承了两个名为 v 的字段，一个来自于其超类 SuperTest，另一个来自于其超接口 Frob。本质上允许这样做，但是由于使用了方法 printV 中的简单名称 v，因此会发生编译时错误：不能确定哪个 v 是想要的。

下面的变体使用字段访问表达式 super.v 引用类 SuperTest 中声明的名为 v 的字段，并使用限定名称 Frob.v 来引用接口 Frob 中声明的名为 v 的字段：

```

interface Frob { float v = 2.0f; }
class SuperTest { int v = 3; }
class Test extends SuperTest implements Frob {
    public static void main(String[] args) {
        new Test().printV();
    }
}

```

```

    void printV() {
        System.out.println((super.v + Frob.v)/2);
    }
}

```

它通过编译并输出：

2.5

即使两个不同的继承字段具有相同的类型、相同的值，并且都是 `final` 类型，通过简单名称指向其中一个字段的任何引用都会被视作是有歧义的，并会导致编译时错误。在下面的示例中：

```

interface Color { int RED=0, GREEN=1, BLUE=2; }
interface TrafficLight { int RED=0, YELLOW=1, GREEN=2; }
class Test implements Color, TrafficLight {
    public static void main(String[] args) {
        System.out.println(GREEN); // compile-time error
        System.out.println(RED);   // compile-time error
    }
}

```

可以理解，指向 `GREEN` 的引用应该被视为是有歧义的，这是由于类 `Test` 继承了 `GREEN` 的两个具有不同值的不同声明。这个示例的要点在于，指向 `RED` 的引用也会被视为是有歧义的，这是由于继承了两个不同的声明。两个名为 `RED` 的字段碰巧具有相同的类型和相同的不变值这一事实不会影响这种判断。

#### 8.3.3.4 示例：字段的重新继承

如果通过多种途径从一个接口继承相同的字段声明，则该字段将被视为只被继承一次。可以通过其简单名称无歧义地引用它。例如，在下面的代码中：

```

public interface Colorable {
    int RED = 0xff0000, GREEN = 0x00ff00, BLUE = 0x0000ff;
}
public interface Paintable extends Colorable {
    int MATTE = 0, GLOSSY = 1;
}
class Point { int x, y; }
class ColoredPoint extends Point implements Colorable {
    . . .
}
class PaintedPoint extends ColoredPoint implements Paintable
{
    . . . RED . . .
}

```

类 `PaintedPoint` 通过其直接超类 `ColoredPoint` 和其直接超接口 `Paintable` 继承字段 `RED`、`GREEN` 和 `BLUE`。然而，可以在类 `PaintedPoint` 内部无歧义地使用简单名称 `RED`、`GREEN` 和 `BLUE`，来引用在接口 `Colorable` 中声明的字段。



## 8.4 方法声明

实际参数和观点的多样性包围了各种方法。

——Michael de Montaigne (1533~1592), 《Of Experience》

方法声明了可被调用的可执行代码，传递固定数量的值作为参数。

*MethodDeclaration:*

*MethodHeader MethodBody*

*MethodHeader:*

*MethodModifiers*<sub>opt</sub> *TypeParameters*<sub>opt</sub> *ResultType* *MethodDeclarator*

*Throws*<sub>opt</sub>

*ResultType:*

*Type*

*void*

*MethodDeclarator:*

*Identifier* ( *FormalParameterList*<sub>opt</sub> )

第 8.4.3 节中描述了 *MethodModifiers*; 第 8.4.4 节中描述了方法的 *TypeParameters* 子句; 第 8.4.6 节描述了 *Throws* 子句; 第 8.4.7 节描述了 *MethodBody*。方法声明指定了方法返回的值类型，或者使用关键字 *void* 来指定方法不返回值。

*MethodDeclarator* 中的 *Identifier* 可用在一个名称中，用于引用方法。类可以声明一个与类或者类的字段、成员类或成员接口同名的方法，但是这是一种糟糕的方式。

为了保持与 Java 平台的旧版本的兼容性，允许返回一个数组的方法的声明形式放置（一些或全部）空括号对，它们构成了参数列表后面的数组类型的声明。这受到过时的产生式支持：

*MethodDeclarator:*

*MethodDeclarator* [ ]

但是不应该将其用在新代码中。

如果方法体把两个具有重写等价的签名（8.4.2 节）（名称、参数数量和任何参数的类型）的方法声明为成员，则会发生编译时错误。方法和字段可以具有相同的名称，因为它们使用在不同的环境中，并且通过不同的查找过程（6.5 节）消除歧义。

### 8.4.1 形参

方法或构造函数的形参（如果有的话）是通过逗号分隔的参数指定符的列表指定的。每个参数指定符包含类型 [可以选择前置 *final* 修饰符和/或一条或多条注释（9.7 节）] 和指定参数名称的标识符（可以选择后接括号）。列表中的最后一个形参是特殊的：它可以

是一个可变元数参数 (*variable arity parameter*), 通过接在类型后面的省略号来指示:

*FormalParameterList:*

*LastFormalParameter*

*FormalParameters* , *LastFormalParameter*

*FormalParameters:*

*FormalParameter*

*FormalParameters* , *FormalParameter*

*FormalParameter:*

*VariableModifiers* *Type* *VariableDeclaratorId*

*VariableModifiers:*

*VariableModifier*

*VariableModifiers* *VariableModifier*

*VariableModifier: one of*

*finalAnnotation*

*LastFormalParameter:*

*VariableModifiers* *Type*...<sub>opt</sub> *VariableDeclaratorId*

*FormalParameter*

下面重复了第 8.3 节的内容, 以使得这里的介绍更清楚:

*VariableDeclaratorId:*

*Identifier*

*VariableDeclaratorId* [ ]

如果一个方法或构造函数没有参数, 则在方法或构造函数的声明中只会出现一对空括号。

如果相同方法或构造函数的两个形参被声明成具有相同的名称 (也就是说, 它们的声明提及相同的 *Identifier*), 那么就会发生编译时错误。

如果关于形参的注释 *a* 对应于一个注释类型 *T*, 并且 *T* 具有 (元) 注释 *m*, 它对应于 *annotation.Target*, 那么 *m* 必须具有一个其值为 *annotation.ElementType.PARAMETER* 的元素, 否则就会发生编译时错误。第 9.7 节中进一步描述了注释修饰符。

如果在方法或构造函数的主体内对声明为 *final* 的方法或构造函数参数赋值, 则会发生编译时错误。

当调用方法或构造函数时 (15.12 节), 在执行方法或构造函数的主体之前, 实参表达式的值会初始化最新创建的参数变量, 即声明的每个 *Type*。出现在 *DeclaratorId* 中的 *Identifier* 可以用作方法或构造函数的主体中的一个简单名称, 用于引用形参。

如果最后一个形参是类型 *T* 的可变元数参数, 则将其视作定义了一个 *T[]* 类型的形参。

这样，方法就是可变元数方法 (*variable arity method*)。否则，它就是固定元数方法 (*fixed arity method*)。可变元数方法的调用可以包含比形参更多的实参表达式。对于和可变元数参数之前的形参不对应的所有实参表达式，将会对其进行求值，并把结果存储在将传递给方法调用 (15.12.4.2 节) 的数组中。

方法 (8.4.1 节) 或构造函数 (8.8.1 节) 的参数作用域是方法或构造函数的整个主体。

这些参数名称不能被重新声明为方法的局部变量，或者方法或构造函数的 `try` 语句中的 `catch` 子句的异常参数。但是，可以在嵌套在方法或构造函数内的类声明内部的任何位置屏蔽方法或构造函数的参数。这样一个嵌套类声明可以声明本地类 (14.3 节) 或匿名类 (15.9 节)。

只使用简单名称来引用形参，永远也不要使用限定名称 (6.6 节) 来引用它。

`float` 类型的方法或构造函数参数总是包含浮点值集 (4.2.3 节) 的元素；类似地，`double` 类型的方法或构造函数参数总是包含双精度值集的元素。`float` 类型的方法或构造函数参数不允许包含并非浮点值集元素的浮点扩展的指数值集的元素，`double` 类型的方法参数也不允许包含并非双精度值集元素的双精度扩展的指数值集的元素。

在对应于参数变量的实参表达式不是精确浮点 (15.4 节) 的地方，对实参表达式求值允许使用从合适的扩展指数值集抽取的中间值。在把这个表达式的结果存储到参数变量之前，通过方法调用转换 (5.3 节) 将其映射到相应的标准值集中最接近的值。

## 8.4.2 方法签名

在一个类中声明两个具有重写等价签名 (定义如下) 的方法，将会发生编译时错误。

如果两个方法具有相同的名称和参数类型，则它们具有相同的签名。

如果下面所有的条件均成立，则两个方法或构造函数声明  $M$  和  $N$  具有相同的参数类型：

- 它们具有相同数量的形参 (可能是零个)。
- 它们具有相同数量的类型参数 (可能是零个)。
- 设  $\langle A_1, \dots, A_n \rangle$  是  $M$  的形式类型参数， $\langle B_1, \dots, B_n \rangle$  是  $N$  的形式类型参数。在将  $N$  的类型中出现的每个  $B_i$  重命名为  $A_i$  之后， $M$  和  $N$  的相应类型变量的界限和参数类型是相同的。

如果以下两个条件之一成立，则方法  $m1$  的签名是方法  $m2$  的签名的子签名：

- $m2$  具有与  $m1$  相同的签名。
- $m1$  的签名与  $m2$  的签名的擦除相同。

### 讨论

这里定义的子签名的概念旨在表达具有不同签名，但是一个可以重写另一个的两个方法之间的关系。

确切地讲，允许其签名未使用泛型类型的方法重写那个方法的任何泛型化版本。这很重要，因为库设计者可以独立于客户自由地泛型化方法，客户则可以定义库的子类或子接口。考虑下面的示例：

```
class CollectionConverter {  
    List toList(Collection c) {...}
```

```

}
class Overrider extends CollectionConverter{
    List toList(Collection c) {...}
}

```

现在，假定该代码是在引入泛型技术之前编写的，现在类 `CollectionConverter` 的作者决定泛型化该代码，从而得到下面的代码：

```

class CollectionConverter {
    <T> List<T> toList(Collection<T> c) {...}
}

```

在不进行特殊分配的情况下，`Overrider.toList()` 将不再重写 `CollectionConverter.toList()`。相反，这段代码将是非法的。这将极大地阻止使用泛型技术，因为库的编写者将为是否迁移现有的代码而踌躇不决。

当且仅当  $m1$  是  $m2$  的子签名或者  $m2$  是  $m1$  的子签名时，两个方法签名  $m1$  和  $m2$  是重写等价的。

下面的示例：

```

class Point implements Move {
    int x, y;
    abstract void move(int dx, int dy);
    void move(int dx, int dy) { x += dx; y += dy; }
}

```

会引发一个编译时错误，这是由于它使用相同的（因此是重写等价的）签名声明了两个 `move` 方法。这是一个错误，即使其中一个签名是 `abstract`。

### 8.4.3 方法修饰符

*MethodModifiers:*

*MethodModifier*

*MethodModifiers MethodModifier*

*MethodModifier: one of*

*Annotation* public protected private abstract static  
final synchronized native strictfp

第 6.6 节讨论了访问修饰符 `public`、`protected` 和 `private`。如果在一个方法声明中相同的修饰符出现了不止一次，或者如果一个方法声明具有不止一个访问修饰符 `public`、`protected` 和 `private`，则会发生编译时错误。如果包含关键字 `abstract` 的方法声明还包含关键字 `private`、`static`、`final`、`native`、`strictfp` 或 `synchronized` 中的任何一个，则会发生编译时错误。如果包含关键字 `native` 的方法声明还包含 `strictfp`，则会发生编译时错误。

如果方法声明上的注释  $a$  对应于一个注释类型  $T$ ，并且  $T$  具有（元）注释  $m$ ，它对应

于 `annotation.Target`，那么 `m` 必须具有一个其值为 `annotation.ElementType.METHOD` 的元素，否则就会发生编译时错误。第 9.7 节中进一步描述了注释。

如果方法声明中出现了两个或多个方法修饰符，那么习惯上让它们出现的顺序与上面所示 `MethodModifier` 的产生式中的顺序一致，尽管不要求如此。

#### 8.4.3.1 abstract 方法

`abstract` 方法声明引入该方法作为成员，并且会提供其签名（8.4.2 节）、返回类型和 `throws` 子句（如果有的话），但是不会提供一个实现。`abstract` 方法 `m` 的声明必须直接出现在 `abstract` 类（称之为 `A`）中，除非它出现在一个枚举（8.9 节）中；否则，就会发生编译时错误。对于 `A` 的每个子类，如果不是 `abstract`，则必须提供 `m` 的实现，否则就会发生编译时错误，如第 8.1.1.1 节中所述。

如果把 `private` 方法声明为 `abstract`，则会发生编译时错误。

让子类实现一个 `private abstract` 方法是不可能的，这是由于 `private` 方法不会被子类继承；因此，永远不能使用这种方法。

把 `static` 方法声明为 `abstract` 会引发编译时错误。

把 `final` 方法声明为 `abstract` 也会引发编译时错误。

`abstract` 类可以通过提供另一个 `abstract` 方法声明来重写一个 `abstract` 方法。

这可以提供一个位置用于放置文档注释，以精炼返回类型，或者声明一组可以被该方法抛出的异常（11.2 节），当它被其子类实现时，这将会更受限制。考虑下面的这段代码：

```
class BufferEmpty extends Exception {
    BufferEmpty() { super(); }
    BufferEmpty(String s) { super(s); }
}
class BufferError extends Exception {
    BufferError() { super(); }
    BufferError(String s) { super(s); }
}
public interface Buffer {
    char get() throws BufferEmpty, BufferError;
}
public abstract class InfiniteBuffer implements Buffer {
    public abstract char get() throws BufferError;
}
```

类 `InfiniteBuffer` 中方法 `get` 的重写声明指出 `InfiniteBuffer` 的任何子类中的方法 `get` 永远不会抛出一个 `BufferEmpty` 异常，根据推断可知，这是由于它会生成缓冲区中的数据，因此永远不会用完数据。

如果一个实例方法不是 `abstract`，则可以被一个 `abstract` 方法重写。

例如，我们可以定义一个 `abstract` 类 `Point`，它需要其子类实现 `toString`（如果它们是完整的、可实例化的类）：

```
abstract class Point {
    int x, y;
```

```

    public abstract String toString();
}

```

toString 的这个 abstract 声明重写了类 Object 的非 abstract 的 toString 方法（类 Object 是类 Point 的隐式直接超类）。添加以下代码：

```

class ColoredPoint extends Point {
    int color;
    public String toString() {
        return super.toString() + ": color " + color; // error
    }
}

```

将导致一个编译时错误，这是由于调用 super.toString() 会引用类 Point 中的方法 toString，它是 abstract，因此不能被调用。仅当类 Point 显式使之可通过某些其他方法使用时，类 Object 的方法 toString 才可供类 ColoredPoint 使用，如以下代码：

```

abstract class Point {
    int x, y;
    public abstract String toString();
    protected String objString() { return super.toString(); }
}
class ColoredPoint extends Point {
    int color;
    public String toString() {
        return objString() + ": color " + color; // correct
    }
}

```

#### 8.4.3.2 static 方法

被声明为 static 的方法称为类方法。类方法总是会被调用，而无需引用特殊的对象。如果试图使用关键字 this 或关键字 super 引用当前对象，或者引用类方法体中任何周围的声明的类型参数，则会导致编译时错误。把 static 方法声明为 abstract 将会导致编译时错误。

未被声明为 static 的方法称为实例方法，有时也称为非 static 方法。实例方法总是关于一个对象被调用，在方法体执行期间，该对象将变成关键字 this 和 super 引用的当前对象。

#### 8.4.3.3 final 方法

可以把方法声明为 final，以防止子类重写或隐藏它。重写或隐藏 final 方法的尝试将会引发编译时错误。

private 方法以及直接在 final 类（8.1.1.2 节）中声明的所有方法的行为方式就像它们是 final 方法一样，这是由于不可能重写它们。

将 final 方法声明为 abstract 将会引发编译时错误。

在运行时，机器代码生成器或优化器可以“内嵌”final 方法体，从而代替用其主体



中的代码调用该方法。内嵌进程必须保存方法调用的语义。特别地，如果实例方法调用的目标为 null，则必须抛出一个 NullPointerException，即使该方法是内嵌的也是如此。编译器必须确保异常将在正确的点被抛出，以便在方法调用之前，查看到按正确的顺序对方法的实参进行了求值。

考虑下面的示例：

```
final class Point {
    int x, y;
    void move(int dx, int dy) { x += dx; y += dy; }
}
class Test {
    public static void main(String[] args) {
        Point[] p = new Point[100];
        for (int i = 0; i < p.length; i++) {
            p[i] = new Point();
            p[i].move(i, p.length-1-i);
        }
    }
}
```

这里，把类 Point 的方法 move 内嵌在方法 main 中，将把 for 循环转换成如下形式：

```
for (int i = 0; i < p.length; i++) {
    p[i] = new Point();
    Point pi = p[i];
    int j = p.length-1-i;
    pi.x += i;
    pi.y += j;
}
```

随后将对该循环进行进一步的优化。

这种内嵌不能在编译时完成，除非能够保证 Test 和 Point 始终会一起重新编译，从而使得无论何时 Point——更确切地讲是其 move 方法——发生变化，Test.main 的代码也会被更新。

#### 8.4.3.4 native 方法

native 方法是用平台相关的代码实现的，这些代码通常是用另一种编程语言（如 C、C++、FORTRAN 或汇编语言）编写的。native 方法体只作为一个分号（而不是块语句）给出，指示实现被省略。

如果将 native 方法声明为 abstract，则会发生编译时错误。

例如，包 java.io 的类 RandomAccessFile 可以声明如下 native 方法：

```
package java.io;
public class RandomAccessFile
    implements DataOutput, DataInput
{
    ...
    public native void open(String name, boolean writeable)
        throws IOException;
}
```

```

    public native int readBytes(byte[] b, int off, int len)
        throws IOException;
    public native void writeBytes(byte[] b, int off, int len)
        throws IOException;
    public native long getFilePointer() throws IOException;
    public native void seek(long pos) throws IOException;
    public native long length() throws IOException;
    public native void close() throws IOException;
}

```

#### 8.4.3.5 strictfp 方法

strictfp 修饰符的作用是使方法体内所有的 float 或 double 表达式都是显式精确浮点的 (15.4 节)。

#### 8.4.3.6 synchronized 方法

synchronized 方法在执行之前会获得一个监视器 (17.1 节)。对于类 (static) 方法, 将使用与该方法的类的 Class 对象关联的监视器。对于实例方法, 将使用与 this (为这个对象调用该方法) 关联的监视器。

这些是可以被 synchronized 语句 (14.19 节) 使用的相同的锁; 因此, 下面的代码:

```

class Test {
    int count;
    synchronized void bump() { count++; }
    static int classCount;
    static synchronized void classBump() {
        classCount++;
    }
}

```

具有与以下代码完全相同的作用:

```

class BumpTest {
    int count;
    void bump() {
        synchronized (this) {
            count++;
        }
    }
    static int classCount;
    static void classBump() {
        try {
            synchronized (Class.forName("BumpTest")) {
                classCount++;
            }
        } catch (ClassNotFoundException e) {
            ...
        }
    }
}

```

更加精心设计的示例：

```
public class Box {
    private Object boxContents;
    public synchronized Object get() {
        Object contents = boxContents;
        boxContents = null;
        return contents;
    }
    public synchronized boolean put(Object contents) {
        if (boxContents != null)
            return false;
        boxContents = contents;
        return true;
    }
}
```

定义了一个设计用于并发使用的类。类 Box 的每个实例都有一个实例变量 boxContents，它可以存储一个指向任何对象的引用。可以通过调用 put 把对象置入 Box 中，如果 Box 已满，则 put 会返回 false。可以通过调用 get 从 Box 中取出某个对象，如果 Box 为空，则 get 会返回一个空引用。

如果 put 和 get 不是 synchronized 方法，并且两个线程正同时为 Box 的同一个实例执行这两个方法，那么代码可能工作不正常。例如，由于同时发生了两个对 put 的调用，导致丢失对某个对象的跟踪。

有关线程和锁的更多讨论，参见第 17 章。

#### 8.4.4 泛型方法

如果一个方法声明了一个或多个类型变量（4.4 节），则它就是泛型方法。这些类型变量被称为该方法的形式类型参数。形式类型参数列表的形式与类或接口的类型参数列表完全相同，如第 8.1.2 节所述。

方法的类型参数的作用域是方法的整个声明，包括类型参数部分本身。因此，类型参数可以作为它们自身界限的一部分出现，或者作为相同部分中声明的其他类型参数的界限出现。

当调用泛型方法时，不必提供泛型方法的类型参数。相反，几乎总是会推断出它们，如第 15.12.2.7 节所述。

#### 8.4.5 方法返回类型

如果方法返回一个值，则方法的返回类型声明了方法返回的值的类型，或者指出方法返回类型为 void。

当且仅当下列条件成立时，返回类型为  $R_1$  的方法声明  $d_1$  是另一个返回类型为  $R_2$  的方法  $d_2$  的可替代的返回类型：

- 如果  $R_1$  是基本类型，则  $R_2$  与  $R_1$  相同。

- 如果  $R_1$  是引用类型，则：
  - ◆  $R_1$  是  $R_2$  的子类型，或者可以通过未经检查的转换（5.1.9 节）将  $R_1$  转换成  $R_2$  的子类型。
  - ◆  $R_1 = |R_2|$ 。
- 如果  $R_1$  是 `void`，则  $R_2$  也是 `void`。

#### 讨论

返回类型可替代性的概念总结了返回类型在相互重写的方法之间可能发生变化的方式。

注意，这个定义支持协变返回（*covariant returns*），即返回类型对于子类型（但仅适用于引用类型）的特殊化。

另请注意：还允许未经检查的转换。这是不合理的，并且无论何时使用它都需要未经检查的警告：允许从非泛型代码迁移到泛型代码是作出的一个特殊许可。

### 8.4.6 方法抛出

`throws` 子句用于声明任何受查异常（11.2 节），这些异常可能来自于方法或构造函数的执行：

*Throws:*

`throws ExceptionTypeList`

*ExceptionTypeList:*

`ExceptionType`

`ExceptionTypeList, ExceptionType`

*ExceptionType:*

`ClassType`

`TypeVariable`

如果 `throws` 子句中提及的任何 `ExceptionType` 不是 `Throwable` 的子类型（4.10 节），则会发生编译时错误。在 `throws` 子句中允许（但不要求）提及其他（未经检查的）异常。

对于每个可能来自于方法或构造函数的主体执行的受查异常，除非在方法或构造函数声明中的 `throws` 子句中提及了该异常类型或其子类型，否则将会发生编译时错误。

声明受查异常的要求允许编译器确保包括了用于处理这些异常情况的代码。由于 `throws` 子句中缺少适当的异常类型，使得方法或构造函数无法把抛出的异常情况处理成受查异常，这通常会导致编译时错误。因此，Java 编程语言鼓励很少出现异常情况的编程风格，但是一旦出现异常情况，就将其以这种方式记录在文档中。

不会以这种方式进行检查的预定义异常是指那些极其难以声明其每一次可能发生的异常，包括：

- 通过类 `Error` 的子类表示的异常（例如，`OutOfMemoryError`），它们会由于虚拟

机中或虚拟机本身的失败而被抛出。这些异常中有许多都是由于连接失败而产生的，并且可能发生在程序执行中不可预知的点。尖端的程序可能仍然希望捕获并尝试从其中一些情况中恢复。

- 通过类 `RuntimeException` 的子类表示的异常（例如，`NullPointerException`），它们来自于运行时完整性检查，并且直接通过程序或在库例程中抛出。在程序中包括足够的信息，以归纳出可以证明不会发生异常的位置的可管理数量，这超出了 Java 编程语言的范围，也许还超出了最新的编程语言的范围。

与被重写的或隐藏的方法相比，重写或隐藏另一个方法（8.4.8 节）的方法（包括实现接口中定义的 `abstract` 方法的方法）也许不能被声明成抛出更多受查异常。

更确切地讲，假定  $B$  是一个类或接口， $A$  是  $B$  的超类或超接口， $B$  中的方法声明  $n$  重写或隐藏了  $A$  中的方法声明  $m$ 。如果  $n$  具有一个提及了任何受查异常类型的 `throws` 子句，那么  $m$  必须具有一个 `throws` 子句，并且对于  $n$  的 `throws` 子句中列出的每个受查异常类型，相同的异常类或其超类型类之一必须出现在  $m$  的 `throws` 子句的擦除中：否则，就会发生编译时错误。

如果  $m$  的未擦除的 `throws` 子句不包含  $n$  的 `throws` 子句中每个异常类型的超类型，则必定会发出一个未经检查的警告。

#### 讨论

有关异常和一个大型示例的更多信息，参见第 11 章。

在 `throws` 列表中允许出现类型变量，即使它们不允许出现在 `catch` 子句中。

```
interface PrivilegedExceptionAction<E extends Exception> {
    void run() throws E;
}
class AccessController {
    public static <E extends Exception>
    Object doPrivileged(PrivilegedExceptionAction<E> action) throws E
    { ... }
}
class Test {
    public static void main(String[] args) {
        try {
            AccessController.doPrivileged(
                new PrivilegedExceptionAction<FileNotFoundException>() {
                    public void run() throws FileNotFoundException
                    { ... delete a file ... }
                });
        } catch (FileNotFoundException f) { ... } // do something
    }
}
```

### 8.4.7 方法体

方法体是实现方法的代码块，或者只是一个分号，用于指示缺少实现。当且仅当方法

是 `abstract` (8.4.3 节) 或 `native` (8.4.3 节) 时, 方法体才必须是一个分号。

*MethodBody:*

*Block*

;

如果方法声明是 `abstract` 或 `native`, 并且其方法体是一个代码块, 则会发生编译时错误。如果方法声明既不是 `abstract` 也不是 `native`, 并且其方法体是一个分号, 则会发生编译时错误。

如果为声明为 `void` 的方法提供了一个实现, 但是该实现不需要可执行代码, 则应该把方法体写成一个块, 它不包含任何语句: “{ }”。

如果方法被声明为 `void`, 则其方法体绝对禁止包含任何带有一个表达式的 `return` 语句 (14.17 节)。

如果方法被声明为有一个返回类型, 则其方法体内的每个 `return` 语句 (14.17 节) 都必须有一个表达式。如果方法体可以正常完成 (14.1 节), 则会发生编译时错误。

换句话说, 带有一个返回类型的方法必须只通过使用提供了值返回的返回语句来返回; 而不允许它“丢弃其方法体的末尾”。

注意, 对于一个方法而言, 具有声明的返回类型但不包含任何返回语句, 是有可能的。下面是一个示例:

```
class DizzyDean {  
    int pitch() { throw new RuntimeException("90 mph?!"); }  
}
```

### 8.4.8 继承、重写和隐藏

类 *C* 从其直接超类和直接超接口继承了超类和超接口的所有非私有方法 (不管是否是 `abstract` 方法), 它们是 `public`、`protected` 或者是在与 *C* 相同的包中被声明为具有默认访问方式, 并且既不能被类 *C* 中的声明重写 (8.4.8.1 节), 也不能被隐藏 (8.4.8.2 节)。

#### 8.4.8.1 重写 (通过实例方法)

当且仅当以下所有条件均成立时, 类 *C* 中声明的实例方法 *m1* 才会重写类 *A* 中声明的另一个实例方法 *m2*:

(1) *C* 是 *A* 的子类。

(2) *m1* 的签名是 *m2* 的签名的子签名 (8.4.2 节)。

(3) 以下条件之一成立:

- ◆ *m2* 是 `public`、`protected`, 或者是在与 *C* 相同的包中被声明为具有默认访问方式。

- ◆ *m1* 重写了方法 *m3*, *m3* 不同于 *m1*, *m3* 不同于 *m2*, 所以 *m3* 重写了 *m2*。

此外, 如果 *m1* 不是 `abstract`, 则称 *m1* 实现了它重写的任何和所有 `abstract` 方法的声明。



**讨论**

如果某个方法中的一个形参具有原生类型，而另一个方法中对应的参数具有参数化类型，则重写方法的签名可能不同于被重写方法的签名。

这些规则允许重写方法的签名不同于被重写方法的签名，以适应预先存在的代码的迁移，从而利用泛型技术。更多分析请参见第 8.4.2 节。

如果一个实例方法重写一个 `static` 方法，则会发生编译时错误。

就这个方面而言，方法的重写不同于方法的隐藏（8.3 节），因为允许实例变量隐藏 `static` 变量。

可以通过使用包含关键字 `super` 的方法调用表达式（15.12 节）来访问被重写的方法。注意，在尝试访问被重写的方法时，使用限定名称或转换到超类类型的强制转换并不是有效的方式；就这个方面而言，方法的重写不同于方法的隐藏。有关这个要点的讨论和示例，参见第 15.12.4.9 节。

`strictfp` 修饰符的存在与否对重写方法和实现 `abstract` 方法的规则完全没有影响。例如，允许非精确浮点的方法重写精确浮点的方法，允许精确浮点的方法重写非精确浮点的方法。

#### 8.4.8.2 隐藏（通过类方法）

如果类声明了一个 `static` 方法  $m$ ，那么就称声明  $m$  隐藏了该类的超类或超接口中的任何方法  $m'$ ，其中  $m$  的签名是  $m'$  的签名的子签名（8.4.2 节），否则， $m'$  对于该类中的代码是可访问的。如果 `static` 方法隐藏了一个实例方法，则会发生编译时错误。

就这个方面而言，方法的隐藏不同于字段的隐藏（8.3 节），因为允许 `static` 变量隐藏实例变量。隐藏也不同于屏蔽（6.3.1 节）和模糊（6.3.2 节）。

可以通过使用限定名称、包含关键字 `super` 或转换到超类类型的强制转换的方法调用表达式（15.12 节），来访问被隐藏的方法。就这个方面而言，方法的隐藏类似于字段的隐藏。

#### 8.4.8.3 重写和隐藏中的要求

如果返回类型为  $R_1$  的方法声明  $d_1$  重写或隐藏了另一个返回类型为  $R_2$  的方法  $d_2$  的声明，那么  $d_1$  必须是  $d_2$  的可替代返回类型，否则会发生编译时错误。此外，如果  $R_1$  不是  $R_2$  的子类型，则必须发出未经检查的警告 [除非禁止这样做（9.6.1.5 节）]。

方法声明绝对禁止具有一个与它所重写或隐藏的任何方法的 `throws` 子句相冲突（8.4.6 节）的 `throws` 子句；否则，就会发生编译时错误。

**讨论**

上述规则允许协变返回类型——在重载某个方法时，精炼该方法的返回类型。例如，下列声明是合法的，尽管它们在 Java 编程语言以前的版本中是非法的：

```
class C implements Cloneable {  
    C copy() { return (C)clone(); }  
}
```

```
class D extends C implements Cloneable {
    D copy() { return (D)clone(); }
}
```

重写放宽的规则还允许程序员放宽关于实现接口的 abstract 类的条件。

### 讨论

考虑下面的代码：

```
class StringSorter {
    // takes a collection of strings and converts it to a sorted list
    List toList(Collection c) {...}
}
```

假定创建 StringCollector 的一个子类：

```
class Overrider extends StringSorter{
    List toList(Collection c) {...}
}
```

现在，在某一点上，StringSorter 的作者决定泛型化代码：

```
class StringSorter {
    // takes a collection of strings and converts it to a list
    List<String> toList(Collection<String> c) {...}
}
```

当相对于 StringSorter 的新定义编译 Overrider 时，将会给出一个未经检查的警告，这是由于 Overrider.toList() 的返回类型是 List，它不是被重写的方法 List<String> 的返回类型的子类型。

就这些方面而言，方法的重写不同于字段的隐藏（8.3 节），因为允许一个字段隐藏另一种类型的字段。

如果类型声明  $T$  具有一个成员方法  $m_1$ ，并且存在  $T$  或其超类型中声明的方法  $m_2$ ，使得以下所有条件均成立，则会发生编译时错误：

- $m_1$  和  $m_2$  具有相同的名称。
- 可以从  $T$  访问  $m_2$ 。
- $m_1$  的签名不是  $m_2$  的签名的子签名（8.4.2 节）。
- $m_1$  或  $m_1$  的某个重写版本（直接或间接）具有与  $m_2$  或  $m_2$  的某个重写版本（直接或间接）相同的擦除。

### 讨论

这些限制是必要的，因为泛型技术是通过擦除实现的。上述规则暗示在相同类中用相同名称声明的方法必须具有不同的擦除。它还暗示类型声明不能实现或扩展相同泛型接口的两个不同的调用。下面是一些进一步的示例。

一个类不能具有两个有相同名称和类型擦除的成员方法。

```
class C<T> { T id (T x) {...} }  
class D extends C<String> {  
    Object id(Object x) {...}  
}
```

这是非法的，因为 `D.id(Object)` 是 `D` 的成员，`C<String>.id(String)` 是在 `D` 的超类型中声明的，并且：

- 两个方法具有相同的名称 `id`
- `C<String>.id(String)` 对于 `D` 是可访问的
- `D.id(Object)` 的签名不是 `C<String>.id(String)` 的签名的子签名
- 两个方法具有相同的擦除

## 讨论

类的两个不同的方法不能重写具有相同擦除的方法。

```
class C<T> { T id (T x) {...} }  
interface I<T> { T id(T x); }  
class D extends C<String> implements I<Integer> {  
    String id(String x) {...}  
    Integer id(Integer x) {...}  
}
```

这也是非法的，因为 `D.id(String)` 是 `D` 的成员，`D.id(Integer)` 是在 `D` 中声明的，并且：

- 两个方法具有相同的名称 `id`。
- 两个方法具有不同的签名。
- `D.id(Integer)` 可以被 `D` 访问。
- `D.id(String)` 重写 `C<String>.id(String)`，`D.id(Integer)` 重写 `I.id(Integer)`，而这两个被重写的方法具有相同的擦除。

重写或隐藏的方法的访问修饰符（6.6 节）必须提供与被重写或被隐藏的方法至少一样的访问权限，否则，就会发生编译时错误。更详细地讲：

- 如果被重写或被隐藏的方法是 `public`，那么重写或隐藏的方法必须是 `public`；否则，就会发生编译时错误。
- 如果被重写或被隐藏的方法是 `protected`，那么重写或隐藏的方法必须是 `protected` 或 `public`；否则，就会发生编译时错误。
- 如果被重写或被隐藏的方法具有默认的（包）访问，那么重写或隐藏的方法绝对禁止是 `private` 否则，就会发生编译时错误。

注意，从这些术语的技术意义上讲，`private` 方法不能被隐藏或被重写。这意味着：子类可以用与其超类之一中的某个 `private` 方法相同的签名来声明一个方法，并且不需要该方法的返回类型或 `throws` 子句与超类中的那个 `private` 方法的返回类型或 `throws`

子句有任何关系。

#### 8.4.8.4 利用重写等价的签名继承方法

一个类利用重写等价的签名(8.4.2节)继承多个方法是可能的。如果类C继承了一个具体的方法,其签名是C继承的另一个具体方法的子签名,则会发生编译时错误。

#### 讨论

如果超类是参数,并且它具有两个泛型声明不同的方法,但是它们在特殊调用中会使用相同的签名,则可能会发生这种情况。

否则,就有如下两种可能的情况:

- 如果被继承的方法之一不是 `abstract`, 则有如下两种子情况:
  - ◆ 如果非 `abstract` 的方法是 `static`, 则会发生编译时错误。
  - ◆ 否则, 非 `abstract` 的方法将会被视作要重写(从而要实现)代表继承它的类的所有其他方法。如果非 `abstract` 的方法的签名不是每个其他被继承的方法的子签名, 则必须发出一个未经检查的警告[除非禁止这样做(9.6.1.5节)]。如果非 `abstract` 方法的返回类型不能替代(8.4.5节)每个其他被继承方法的返回类型, 那么还会发生编译时错误。如果非 `abstract` 方法的返回类型不是任何其他被继承方法的返回类型的子类型, 则必须发出一个未经检查的警告。此外, 如果被继承的非 `abstract` 方法具有一个 `throws` 子句, 该子句与任何被继承的方法的 `throws` 子句相冲突(8.4.6节), 则会发生编译时错误。
- 如果所有被继承的方法都是 `abstract` 方法, 那么类就必须是一个 `abstract` 类, 并且会被视作继承所有的 `abstract` 方法。对于其中任意两个被继承的方法来说, 如果其中一个方法的返回类型不能代替另一个方法的返回类型, 则会发生编译时错误(在这种情况下, `throws` 子句不会引发错误)。

可以通过多条途径从一个接口继承相同的方法声明。这个事实不会引起难以解决的情况, 并且其自身永远不会导致编译时错误。

### 8.4.9 重载

如果类的两个方法(无论它们是在同一个类中声明的, 或者都是被类继承的, 抑或一个是声明的另一个是继承的)具有相同的名称, 但是其签名不是重写等价的, 那么就称方法名称是被重载的。这个事实不会引起难以解决的情况, 并且其自身永远不会导致编译时错误。在具有相同名称的两个方法的返回类型之间或者 `throws` 子句之间不需要有任何关系, 除非它们的签名是重写等价的。

方法是在逐个签名的基础上进行重写的。

例如, 如果类声明了两个具有相同名称的 `public` 方法, 并且一个子类重写了其中一个方法, 则该子类仍会继承另一个方法。

当调用一个方法(15.12节)时, 在编译时要使用实参(以及任何显式类型参数)的

数量和参数的编译时类型来确定将被调用的方法的签名（15.12.2 节）。如果被调用的方法是一个实例方法，则要被调用的实际方法将在运行时使用动态方法查找（15.12.4 节）确定。

### 8.4.10 方法声明的示例

下面的示例阐释了有关方法声明的一些（可能是微妙的）要点。

#### 8.4.10.1 示例：重写

在下面的示例中：

```
class Point {
    int x = 0, y = 0;
    void move(int dx, int dy) { x += dx; y += dy; }
}
class SlowPoint extends Point {
    int xLimit, yLimit;
    void move(int dx, int dy) {
        super.move(limit(dx, xLimit), limit(dy, yLimit));
    }
    static int limit(int d, int limit) {
        return d > limit ? limit : d < -limit ? -limit : d;
    }
}
```

类 `SlowPoint` 用其自己的 `move` 方法重写了类 `Point` 的 `move` 方法的声明，这限制了在每次调用方法时点可以移动的距离。当为类 `SlowPoint` 的实例调用 `move` 方法时，始终会调用类 `SlowPoint` 中的重写定义，即使从类型为 `Point` 的变量获得一个指向 `SlowPoint` 对象的引用也是如此。

#### 8.4.10.2 示例：重载、重写和隐藏

在下面的示例中：

```
class Point {
    int x = 0, y = 0;
    void move(int dx, int dy) { x += dx; y += dy; }
    int color;
}
class RealPoint extends Point {
    float x = 0.0f, y = 0.0f;
    void move(int dx, int dy) { move((float)dx, (float)dy); }
    void move(float dx, float dy) { x += dx; y += dy; }
}
```

类 `RealPoint` 用它自己的 `float` 实例变量 `x` 和 `y` 隐藏了类 `Point` 的 `int` 实例变量 `x` 和 `y` 的声明，并用它自己的 `move` 方法重写了类 `Point` 的 `move` 方法。它还用另一个具有不同签名（8.4.2 节）的方法重载了名称 `move`。在本示例中，类 `RealPoint` 的成

员包括从类 `Point` 继承的实例变量 `color`、`RealPoint` 中声明的 `float` 实例变量 `x` 和 `y`，以及 `RealPoint` 中声明的两个 `move` 方法。要为任何特殊的方法调用选择类 `RealPoint` 的哪个重载的 `move` 方法，这是在编译时通过第 15.12 节中描述的重载解决过程确定的。

#### 8.4.10.3 示例：不正确的重写

本示例是前一节中那个示例的扩展变体：

```
class Point {
    int x = 0, y = 0, color;
    void move(int dx, int dy) { x += dx; y += dy; }
    int getX() { return x; }
    int getY() { return y; }
}
class RealPoint extends Point {
    float x = 0.0f, y = 0.0f;
    void move(int dx, int dy) { move((float)dx, (float)dy); }
    void move(float dx, float dy) { x += dx; y += dy; }
    float getX() { return x; }
    float getY() { return y; }
}
```

其中，类 `Point` 提供了方法 `getX` 和 `getY`，它们返回其字段 `x` 和 `y` 的值；然后，类 `RealPoint` 通过用相同的签名声明方法重写了这些方法。这导致在编译时会发生两个错误，每个方法一个错误，这是由于返回类型不匹配；类 `Point` 中的方法会返回 `int` 类型的值，而类 `RealPoint` 中对应的重写方法则会返回 `float` 类型的值。

#### 8.4.10.4 示例：重写与隐藏

本示例校正了前一节中那个示例的错误：

```
class Point {
    int x = 0, y = 0;
    void move(int dx, int dy) { x += dx; y += dy; }
    int getX() { return x; }
    int getY() { return y; }
    int color;
}
class RealPoint extends Point {
    float x = 0.0f, y = 0.0f;
    void move(int dx, int dy) { move((float)dx, (float)dy); }
    void move(float dx, float dy) { x += dx; y += dy; }
    int getX() { return (int)Math.floor(x); }
    int getY() { return (int)Math.floor(y); }
}
```

其中，类 `RealPoint` 中的重写方法 `getX` 和 `getY` 具有与它们重写的类 `Point` 中的方法相同的返回类型，因此，这段代码可以成功地通过编译。

然后，考虑下面这个测试程序：



```
class Test {
    public static void main(String[] args) {
        RealPoint rp = new RealPoint();
        Point p = rp;
        rp.move(1.71828f, 4.14159f);
        p.move(1, -1);
        show(p.x, p.y);
        show(rp.x, rp.y);
        show(p.getX(), p.getY());
        show(rp.getX(), rp.getY());
    }
    static void show(int x, int y) {
        System.out.println("(" + x + ", " + y + ")");
    }
    static void show(float x, float y) {
        System.out.println("(" + x + ", " + y + ")");
    }
}
```

该程序的输出如下：

```
(0, 0)
(2.7182798, 3.14159)
(2, 3)
(2, 3)
```

输出的第一行说明了如下事实：RealPoint 的实例实际上包含类 Point 中声明的两个整型字段；也就是说，它们的名称对于类 RealPoint 的声明内出现的代码是隐藏的（并且对于任何子类的声明也是如此）。当使用一个指向 Point 类型的变量中的 RealPoint 类实例的引用来访问字段 x 时，将会访问类 Point 中声明的整型字段 x。其值为 0 的事实表明方法调用 p.move(1, -1) 不会调用类 Point 的方法 move；相反，它会调用类 RealPoint 的重载方法 move。

输出的第二行表明字段访问 rp.x 引用类 RealPoint 中声明的字段 x。该字段是 float 类型，因此输出的这个第二行会显示浮点值。顺便说一句，这还说明了方法名 show 被重载的事实；方法调用中的参数的类型指定将会调用两个定义中的哪一个定义。

输出的后两行表明方法调用 p.getX() 和 rp.getX() 中的每一个都会调用类 RealPoint 中声明的 getX 方法。实际上，不管我们用于存储指向对象的引用的变量类型是什么，都无法从 RealPoint 类体的外部为类 RealPoint 的实例调用类 Point 的 getX 方法。因此，我们看到字段和方法的行为方式不同：隐藏不同于重写。

#### 8.4.10.5 示例：被隐藏类方法的调用

可以使用一个引用来调用被隐藏的类（static）方法，该引用的类型是实际包含该方法的声明的类。就这一方面而言，静态方法的隐藏不同于实例方法的重写。下面的示例：

```
class Super {
    static String greeting() { return "Goodnight"; }
    String name() { return "Richard"; }
}
```

```

class Sub extends Super {
    static String greeting() { return "Hello"; }
    String name() { return "Dick"; }
}
class Test {
    public static void main(String[] args) {
        Super s = new Sub();
        System.out.println(s.greeting() + ", " + s.name());
    }
}

```

产生如下输出：

```
Goodnight, Dick
```

这是由于 `greeting` 的调用使用 `s` 的类型（即 `Super`）在编译时断定要调用哪个类方法，而 `name` 的调用则使用 `s` 的类（即 `Sub`）在运行时断定要调用哪个实例方法。

#### 8.4.10.6 重写的示例

重写使得子类易于扩展现有类的行为，如本示例所示：

```

import java.io.OutputStream;
import java.io.IOException;
class BufferOutput {
    private OutputStream o;
    BufferOutput(OutputStream o) { this.o = o; }
    protected byte[] buf = new byte[512];
    protected int pos = 0;
    public void putchar(char c) throws IOException {
        if (pos == buf.length)
            flush();
        buf[pos++] = (byte)c;
    }
    public void putstr(String s) throws IOException {
        for (int i = 0; i < s.length(); i++)
            putchar(s.charAt(i));
    }
    public void flush() throws IOException {
        o.write(buf, 0, pos);
        pos = 0;
    }
}
class LineBufferOutput extends BufferOutput {
    LineBufferOutput(OutputStream o) { super(o); }
    public void putchar(char c) throws IOException {
        super.putchar(c);
        if (c == '\n')
            flush();
    }
}
class Test {
    public static void main(String[] args)

```

```

        throws IOException
    {
        LineBufferOutput lbo =
            new LineBufferOutput(System.out);
        lbo.putstr("lbo\nlbo");
        System.out.print("print\n");
        lbo.putstr("\n");
    }
}

```

本示例产生如下输出：

```

lbo
print
lbo

```

类 `BufferOutput` 实现了 `OutputStream` 的一个非常简单的缓冲版本，当缓冲区已满或者调用 `flush` 时，就会刷新输出。

子类 `LineBufferOutput` 只声明了一个构造函数和单独一个方法 `putchar`，该方法重写了 `BufferOutput` 的方法 `putchar`。它从类 `BufferOutput` 继承了方法 `putstr` 和 `flush`。

在 `LineBufferOutput` 对象的 `putchar` 方法中，如果字符参数是换行符，则它会调用 `flush` 方法。关于本示例中重写的关键要点是：在类 `BufferOutput` 中声明的方法 `putstr` 会调用通过当前对象 `this` 定义的 `putchar` 方法，它不必是在类 `BufferOutput` 中声明的 `putchar` 方法。

因此，当在 `main` 中使用 `LineBufferOutput` 对象 `lbo` 调用 `putstr` 时，`putstr` 方法体中的 `putchar` 调用是对象 `lbo` 的 `putchar` 调用，它是检查换行符的 `putchar` 的重写声明。这允许 `BufferOutput` 的子类改变 `putstr` 方法的行为，而无需重新定义它。

有关设计用于扩展的类（如 `BufferOutput`）的文档应该清楚指定类与其子类之间的契约是什么，并且应该清楚指定子类可以用这种方式重写 `putchar` 方法。`BufferOutput` 类的实现者将不希望更改 `BufferOutput` 的将来实现中的 `putstr` 的实现，因此，不希望使用方法 `putchar`，这是由于这将会中断与子类之间预先存在的契约。参见第 13 章（特别是第 13.2 节）中有关二进制兼容性的进一步讨论。

#### 8.4.10.7 示例：由于 *throws* 而导致不正确的重写

本示例在类 `BadPointException` 的声明中使用通常的、传统的形式来声明一种新的异常类型：

```

class BadPointException extends Exception {
    BadPointException() { super(); }
    BadPointException(String s) { super(s); }
}
class Point {
    int x, y;
    void move(int dx, int dy) { x += dx; y += dy; }
}
class CheckedPoint extends Point {
    void move(int dx, int dy) throws BadPointException {

```

```
        if ((x + dx) < 0 || (y + dy) < 0)
            throw new BadPointException();
        x += dx; y += dy;
    }
}
```

本示例会产生一个编译时错误,这是由于类 `CheckedPoint` 中的方法 `move` 的重写声明对于类 `Point` 中未声明的 `move`, 它将抛出一个受查异常。如果这未被视作一个错误,那么如果抛出这个异常,则类型 `Point` 的引用上的方法 `move` 的调用者可以发现它与 `Point` 之间的契约已中断。

删除 `throws` 子句不会有什么帮助:

```
class CheckedPoint extends Point {
    void move(int dx, int dy) {
        if ((x + dx) < 0 || (y + dy) < 0)
            throw new BadPointException();
        x += dx; y += dy;
    }
}
```

现在将会发生一个不同的编译时错误,这是由于 `move` 方法体不能抛出一个受查异常,即 `BadPointException`, 它不会出现在针对 `move` 的 `throws` 子句中。

## 8.5 成员类型声明

成员类是指其声明直接封闭在另一个类或接口声明中的类。类似地,成员接口是指其声明直接封闭在另一个类或接口声明中的接口。第 8.1.6 节中详细说明了成员类或接口的作用域(6.3 节)。

如果类声明了具有某个名称的成员类型,那么就称该类型的声明隐藏了那个类的超类和超接口中具有相同名称的任何和所有可访问的成员类型声明。

在类 `C` 中,名为 `n` 的成员类型的声明 `d` 屏蔽了名为 `n` 的任何其他类型的声明,这些类型在 `d` 出现的那一点上位于作用域内。

如果用简单名称 `C` 声明的成员类或接口直接封闭在具有完全限定名称 `N` 的类的声明中,那么成员类或接口就具有完全限定名称 `N.C`。类从其直接超类和直接超接口继承了超类和超接口的所有非私有成员类型,这些超类和超接口对于类的代码都是可访问的,并且不会被类中的声明隐藏。

类可以从两个接口或者从其超类和一个接口继承两个或多个具有相同名称的类型声明。如果试图通过其简单名称引用任何有歧义地继承的类或接口,则会发生编译时错误。

如果通过多条途径从一个接口继承相同的类型声明,则会将类或接口视作只被继承一次。可以通过其简单名称无歧义地引用它。

### 8.5.1 修饰符

第 6.6 节中讨论了访问修饰符 `public`、`protected` 和 `private`。如果某个成员类

型声明具有多个访问修饰符 `public`、`protected` 和 `private`，则会发生编译时错误。

如同任何类型或成员声明一样，成员类型声明可以具有注释修饰符。

### 8.5.2 静态成员类型声明

`static` 关键字可以修饰非内部类 `T` 的主体内成员类型 `C` 的声明。其作用是声明 `C` 不是一个内部类。就像 `T` 的静态方法在其方法体中不具有 `T` 的当前实例一样，`C` 也不具有 `T` 的当前实例，并且它还不具有任何词汇封闭的实例。

如果一个 `static` 类包含对封闭类的非 `static` 成员的使用，则会发生编译时错误。

成员接口隐式地始终是 `static` 类型。允许（但不要求）成员接口的声明显式地列出 `static` 修饰符。

## 8.6 实例初始化语句

如第 8.8.7.1 节中所指定的那样，在创建类的实例（15.9 节）时，就执行类中声明的实例初始化语句。

*InstanceInitializer:*

*Block*

如果命名类的实例初始化语句可以抛出一个受查异常，则会发生编译时错误，除非在该类的每个构造函数的 `throws` 子句中显式声明了该异常或其子类型之一，并且那个类至少具有一个显式声明的构造函数。匿名类（15.9.5 节）中的实例初始化语句可以抛出任何异常。

上述规则在命名类和匿名类的实例初始化语句之间有所区别。这种区别是精心设计的。给定的匿名类只会在程序中的单独一个点上被实例化，因此，有可能把关于匿名类的实例初始化语句可能引发什么异常的信息直接传播给周围的表达式。另一方面，命名类可以在许多位置进行实例化，因此，传播关于命名类的实例初始化语句可能引发什么异常的信息的惟一方式是通过其构造函数的 `throws` 子句。它遵从一条更自由的规则，该规则也可以用在匿名类中。类似的注释适用于实例变量初始化语句。

如果实例初始化语句不能正常完成（14.21 节），则会发生编译时错误。如果 `return` 语句（14.17 节）出现在实例初始化语句内的任何位置，则会发生编译时错误。

有时会限制在声明之前使用实例变量，即使这些实例变量都在作用域中。有关管理指向实例变量的向前引用的确切规则，参见第 8.3.2.3 节。

允许实例初始化语句引用当前对象 `this`（15.8.3），引用作用域中的任何类型变量（4.4 节），以及使用关键字 `super`（15.11.2 节、15.12 节）。

## 8.7 静态初始化语句

在初始化类时，会执行类中声明的任何静态初始化语句，同时可能使用针对类变量的

任何字段初始化语句（8.3.2 节）来初始化类的类变量（12.4 节）。

*StaticInitializer:*

*static Block*

如果静态初始化语句能够利用受查异常（11.2 节）突然地完成（14.1 节、15.6 节），则会发生编译时错误。如果静态初始化语句不能正常完成（14.21 节），则会发生编译时错误。

静态初始化语句和类变量初始化语句是按代码顺序执行的。

有时会限制在声明之前使用类变量，即使这些类变量都在作用域中。有关管理指向类变量的向前引用的确切规则，参见第 8.3.2.3 节。

如果 `return` 语句（14.17 节）出现在静态初始化语句内的任何位置，则会发生编译时错误。

如果关键字 `this`（15.8.3 节）、在初始化语句外部定义的任何类型变量（4.4 节）或者关键字 `super`（15.11 节、15.12 节）出现在静态初始化语句内的任何位置，则会发生编译时错误。

## 8.8 构造函数声明

铰盘、桥梁、码头、船头、浮舟的建造者面朝着海发呆……  
——沃尔特·惠特曼，《阔斧之歌》（1856）

构造函数用于创建对象，对象是类的实例：

*ConstructorDeclaration:*

*ConstructorModifiers<sub>opt</sub> ConstructorDeclarator*

*Throws<sub>opt</sub> ConstructorBody*

*ConstructorDeclarator:*

*TypeParameters<sub>opt</sub> SimpleTypeName ( FormalParameterList<sub>opt</sub> )*

*ConstructorDeclarator* 中的 *SimpleTypeName* 必须是包含构造函数声明的类的简单名称；否则，就会发生编译时错误。就所有其他方面而言，构造函数声明类似于没有返回类型的方法声明。

下面是一个简单的示例：

```
class Point {
    int x, y;
    Point(int x, int y) { this.x = x; this.y = y; }
}
```

构造函数是通过类实例创建表达式（15.9 节）、字符串串接运算符+（15.18.1 节）引起的转换和串联，以及通过其他构造函数的显式构造函数调用（8.8.7 节）来调用的。构造函



数永远不会通过方法调用表达式（15.12 节）来调用。

对构造函数的访问受到访问修饰符（6.6 节）的管制。

这是有用的，例如，通过声明不可访问的构造函数（8.8.10 节）来防止实例化。

构造函数声明不是成员。它们永远不会被继承，因此不会受到隐藏或重写。

### 8.8.1 形参和形式类型参数

构造函数的形参和形式类型参数在结构和行为方面与方法形参（8.4.1 节）相同。

### 8.8.2 构造函数签名

在一个类中声明两个具有重写等价（8.4.2 节）签名的构造函数会发生编译时错误。在一个类中声明两个其签名有相同擦除（4.6 节）的构造函数也会发生编译时错误。

### 8.8.3 构造函数修饰符

*ConstructorModifiers:*

*ConstructorModifier*

*ConstructorModifiers ConstructorModifier*

*ConstructorModifier: one of*

*Annotation* public protected private

第 6.6 节讨论了访问修饰符 `public`、`protected` 和 `private`。如果在一个构造函数声明中相同的修饰符出现了不止一次，或者如果一个构造函数声明具有不止一个访问修饰符 `public`、`protected` 和 `private`，则会发生编译时错误。

如果没有为普通类的构造函数指定任何访问修饰符，则构造函数具有默认的访问方式。如果没有为枚举类型的构造函数指定任何访问修饰符，则构造函数为 `private`。如果枚举类型（8.9 节）的构造函数被声明为 `public` 或 `protected`，则会发生编译时错误。

如果构造函数上的注释 `a` 对应于一个注释类型 `T`，并且 `T` 具有（元）注释 `m`，它对应于 `annotation.Target`，那么 `m` 必须具有一个其值为 `annotation.ElementType.CONSTRUCTOR` 的元素，否则就会发生编译时错误。第 9.7 节中进一步讨论了注释。

与方法不同的是，构造函数不能是 `abstract`、`static`、`final`、`native`、`strictfp` 或 `synchronized`。构造函数不会被继承，因此无需将其声明为 `final`，并且 `abstract` 构造函数永远不会被实现。构造函数总是关于某个对象被调用，因此把构造函数声明为 `static` 没有意义。对于把构造函数声明为 `synchronized`，没有实际的需要，因为它会把对象锁定在构造之下，直到针对该对象的所有构造函数都完成了它们的工作，它通常才可供其他线程使用。不包括 `native` 构造函数是一种随意的语言设计选择，这使得 Java 虚拟机的实现易于验证，在对象创建期间总会正确地调用超类构造函数。

注意，不能把 *ConstructorModifier* 声明为 `strictfp`。*ConstructorModifier* 和 *MethodModifier*（8.4.3 节）的定义中的这种区别是一种有意的语言设计选择；它有效地确

保当且仅当构造函数的类是精确浮点的时，构造函数才是精确浮点的（15.4节）。

### 8.8.4 泛型构造函数

把构造函数声明为泛型是可能的，这与在其中声明构造函数的类自身是否为泛型无关。如果构造函数声明了一个或多个类型变量（4.4节），则它就是泛型。这些类型变量被称为构造函数的形式类型参数。形式类型参数列表的形式与泛型类或接口的类型参数列表相同，如第8.1.2节所述。

构造函数的类型参数的作用域是构造函数的整个声明，包括类型参数部分本身。因此，类型参数可以作为它们自身界限的一部分出现，或者作为相同部分中声明的其他类型参数的界限出现。

当调用泛型构造函数时，不需要显式提供泛型构造函数的类型参数。当未提供它们时，将会按第15.12.2.7节中所指定的那样推断它们。

### 8.8.5 构造函数 throws

构造函数的 throws 子句在结构和行为方面与方法的 throws 子句（8.4.6节）相同。

### 8.8.6 构造函数的类型

构造函数的类型包含它的签名，以及其 throws 子句中给定的异常类型。

### 8.8.7 构造函数体

构造函数体的第一条语句可以是同一个类或直接超类（8.8.7.1节）的另一个构造函数的显式调用。

*ConstructorBody:*

*{ ExplicitConstructorInvocation<sub>opt</sub> BlockStatements<sub>opt</sub> }*

如果构造函数通过一系列涉及 this 的一个或多个显式构造函数调用来直接或间接地调用它自身，则会发生编译时错误。如果该构造函数是枚举类型（8.9节）的构造函数，那么如果它显式地调用超类构造函数，则会发生编译时错误。

如果构造函数体不是开始于一个显式构造函数调用，并且被声明的构造函数不是根类 Object 的一部分，那么编译器就会隐式地假定构造函数体开始于超类构造函数调用“super();”，即调用其直接超类的无参构造函数。

除了可以进行显式构造函数调用之外，构造函数体类似于方法体（8.4.7节）。如果构造函数体中不包括表达式，则可以在其中使用一条 return 语句（14.17节）。

在下面的示例中：

```
class Point {
    int x, y;
    Point(int x, int y) { this.x = x; this.y = y; }
```

```

    }
    class ColoredPoint extends Point {
        static final int WHITE = 0, BLACK = 1;
        int color;
        ColoredPoint(int x, int y) {
            this(x, y, WHITE);
        }
        ColoredPoint(int x, int y, int color) {
            super(x, y);
            this.color = color;
        }
    }
}

```

ColoredPoint 的第一个构造函数调用第二个构造函数，从而提供一个额外的参数；ColoredPoint 的第二个构造函数调用其超类 Point 的构造函数，并沿着坐标传递。

第 12.5 节和第 15.9 节描述了新的类实例的创建和初始化。

#### 显式构造函数调用

##### *ExplicitConstructorInvocation:*

```

NonWildTypeArgumentsopt this ( ArgumentListopt );
NonWildTypeArgumentsopt super ( ArgumentListopt );
Primary. NonWildTypeArgumentsopt super ( ArgumentListopt );

```

##### *NonWildTypeArguments:*

*< ReferenceTypeList >*

##### *ReferenceTypeList:*

*ReferenceType*  
*ReferenceTypeList, ReferenceType*

显式构造函数调用语句可以分为以下两类：

- **可选构造函数调用**开始于关键字 `this`（可能开始于显式类型参数）。它们用于调用同一个类的可选构造函数。
- **超类构造函数调用**开始于关键字 `super`（可能开始于显式类型参数）或主表达式。它们用于调用直接超类的构造函数。超类构造函数调用可以进一步细分为：
  - ◆ **非限定性超类构造函数调用**开始于关键字 `super`（可能开始于显式类型参数）。
  - ◆ **限定性超类构造函数调用**开始于主表达式。它们允许子类构造函数显式指定最新创建的对象关于直接超类（8.1.3 节）的直接封闭实例。当超类是一个内部类时，这可能是必要的。

下面是限定性超类构造函数调用的一个示例：

```

class Outer {
    class Inner()
}
class ChildOfInner extends Outer.Inner {

```

```
ChildOfInner(){(new Outer()).super();}
}
```

构造函数体中的显式构造函数调用语句不能引用该类或任何超类中声明的任何实例变量或实例方法，或者在任何表达式中使用 `this` 或 `super`；否则，就会发生编译时错误。

例如，如果把上述示例中的 `ColoredPoint` 的第一个构造函数更改为：

```
ColoredPoint(int x, int y) {
    this(x, y, color);
}
```

则会发生一个编译时错误，这是由于不能在超类构造函数调用内使用实例变量。当且仅当下列条件之一成立时，显式构造函数调用语句可以抛出一个异常类型 *E*：

- 构造函数调用的参数列表的某个子表达式可以抛出 *E*。
- *E* 是在被调用的构造函数的 `throws` 子句中声明的。

如果一个匿名类实例创建表达式出现在显式构造函数调用语句内，那么该匿名类不能引用其构造函数被调用的类的任何封闭实例。

例如：

```
class Top {
    int x;
    class Dummy {
        Dummy(Object o) {}
    }
    class Inside extends Dummy {
        Inside() {
            super(new Object() { int r = x; }); // error
        }
        Inside(final int y) {
            super(new Object() { int r = y; }); // correct
        }
    }
}
```

设 *C* 是正被实例化的类，*S* 是 *C* 的直接超类，并且设 *i* 是正被创建的实例。显式构造函数调用的求值过程如下：

- 首先，如果构造函数调用语句是超类构造函数调用，那么必须确定 *i* 关于 *S*（如果有的话）的直接封闭实例。*i* 是否具有关于 *S* 的直接封闭实例，这是由超类构造函数调用确定的，如下：
  - ◆ 如果 *S* 不是一个内部类，或者如果 *S* 的声明出现在静态上下文中，则不存在 *i* 关于 *S* 的直接封闭实例。如果超类构造函数调用是限定性超类构造函数调用，则会发生编译时错误。
  - ◆ 否则：
    - ◇ 如果超类构造函数调用是限定的，那么将会对紧接在 `".super"` 前面的主表达式 *p* 求值。如果主表达式求值为 `null`，则会引发一个 `NullPointerException`，

并且超类构造函数调用会突然完成。否则，这个求值的结果是  $i$  关于  $s$  的直接封闭实例。设  $o$  是  $s$  的直接词汇封闭类；如果  $p$  的类型不是  $o$  或  $o$  的子类，则会发生编译时错误。

◇ 否则：

- 如果  $s$  是本地类（14.3 节），则设  $o$  是  $s$  的最内层词汇封闭类。设  $n$  是一个整数，满足  $o$  是  $c$  的第  $n$  个词汇封闭类。 $i$  关于  $s$  的直接封闭实例是  $this$  的第  $n$  个词汇封闭实例。
- 否则， $s$  是一个内部成员类（8.5 节）。如果  $s$  不是词汇封闭类的成员，或者不是其超类或超接口的成员，则会发生编译时错误。设  $o$  是最内层的词汇封闭类， $s$  是它的一个成员，并且设  $n$  是一个整数，满足  $o$  是  $c$  的第  $n$  个词汇封闭类。 $i$  关于  $s$  的直接封闭实例是  $this$  的第  $n$  个词汇封闭实例。

- 其次，将按照普通方法调用中所做的那样，从左到右对构造函数的参数求值。
- 接下来，调用该构造函数。
- 最后，如果构造函数调用语句是超类构造函数调用，并且构造函数调用语句正常完成，则会执行  $c$  的所有实例变量初始化语句以及  $c$  的所有实例初始化语句。如果一条实例初始化语句或实例变量初始化语句  $i$  在代码中的位置位于另一条实例初始化语句或实例变量初始化语句  $j$  之前，则会在  $j$  之前执行  $i$ 。在执行这个动作时，不考虑超类构造函数调用实际上是作为显式构造函数调用语句出现的，还是自动提供的。可选构造函数调用不会执行这个额外的隐式动作。

### 8.8.8 构造函数重载

构造函数重载与方法重载的行为方式是一样的。重载是在编译时通过每个类实例创建表达式（15.9 节）解决的。

### 8.8.9 默认构造函数

如果一个类不包含任何构造函数声明，则会自动提供一个无参的默认构造函数：

- 如果被声明的类是根类 `Object`，则默认构造函数具有空的主体。
- 否则，默认构造函数将不带参数，并且只会调用无参的超类构造函数。

如果默认构造函数是由编译器提供的，但是超类不具有可访问的无参构造函数，则会发生编译时错误。

默认构造函数没有 `throws` 子句。

如果超类的无参（`nullary`）构造函数具有 `throws` 子句，则会发生编译时错误。

在枚举类型（8.9 节）中，默认构造函数隐式地为 `private`。否则，如果类被声明为 `public`，则会隐式地把访问修饰符 `public`（6.6 节）赋予默认构造函数；如果类被声明为 `protected`，则会隐式地把访问修饰符 `protected`（6.6 节）赋予默认构造函数；如果类被声明为 `private`，则会隐式地把访问修饰符 `private`（6.6 节）赋予默认构造函数；

否则，默认构造函数具有默认的访问方式，它通过不带任何访问修饰符来暗示这一点。

因此，下面的示例：

```
public class Point {  
    int x, y;  
}
```

等价于以下声明：

```
public class Point {  
    int x, y;  
    public Point() { super(); }  
}
```

其中默认构造函数是 `public`，这是由于类 `Point` 是 `public`。

一条简单、直观的规则是：类的默认构造函数具有与类本身相同的访问修饰符。但是注意：这并不暗示无论何时类是可访问的，构造函数就是可访问的。考虑下面的示例：

```
package p1;  
public class Outer {  
    protected class Inner{}  
}  
package p2;  
class SonOfOuter extends p1.Outer {  
    void foo() {  
        new Inner(); // compile-time access error  
    }  
}
```

`Inner` 的构造函数是受保护的。无论如何，该构造函数相对于 `Inner` 是受保护的，而 `Inner` 相对于 `Outer` 是受保护的。因此，`Inner` 在 `SonOfOuter` 中是可访问的，因为它是 `Outer` 的子类。`Inner` 的构造函数在 `SonOfOuter` 中是不可访问的，因为类 `SonOfOuter` 不是 `Inner` 的子类！因此，即使 `Inner` 是可访问的，其默认构造函数也不是可访问的。

### 8.8.10 防止类的实例化

可以把类设计成防止类声明外部的代码通过声明至少一个构造函数来创建类的实例，防止创建隐式的构造函数，以及防止把所有的构造函数都声明为 `private`。`public` 类同样可以防止通过声明至少一个构造函数来创建其包外部的实例，防止创建具有 `public` 访问的默认构造函数，以及防止没有构造函数被声明为 `public`。

因此，在下面的示例中：

```
class ClassOnly {  
    private ClassOnly() { }  
    static String just = "only the lonely";  
}
```

类 `ClassOnly` 不能被实例化，而在下面的示例中：



```
package just;
public class PackageOnly {
    PackageOnly() { }
    String[] justDesserts = { "cheesecake", "ice cream" };
}
```

类 `PackageOnly` 只能在声明它的包 `just` 内被实例化。

## 8.9 枚举

枚举声明的形式如下：

*EnumDeclaration:*

*ClassModifiers*<sub>opt</sub> **enum** *Identifier* *Interfaces*<sub>opt</sub> *EnumBody*

*EnumBody:*

{ *EnumConstants*<sub>opt</sub> , *EnumBodyDeclarations*<sub>opt</sub> }

枚举类型的主体可以包含枚举常量。枚举常量定义了枚举类型的实例。除了通过其枚举常量定义的那些实例外，枚举类型没有其他实例。

### 讨论

如果试图显式实例化一个枚举类型（15.9.1 节），则会发生编译时错误。枚举中的 `final` `clone` 方法确保枚举常量永远不能被复制，并且串行化机制所做的特殊处理确保反串行化永远不会创建复制的实例。禁止枚举类型的自反实例化。这 4 种机制结合在一起确保除了通过枚举常量定义的那些实例之外，不会存在其他的枚举类型的实例。

由于每个枚举常量只有惟一一个实例，因此在比较两个对象引用时，如果已知它们中至少有一个引用的是一个枚举常量，则可以使用 `==` 运算符代替 `equals` 方法（枚举中的 `equals` 方法是 `final` 方法，它只会在其参数上调用 `super.equals` 并返回结果，从而执行同一性比较）。

*EnumConstants:*

*EnumConstant*

*EnumConstants* , *EnumConstant*

*EnumConstant:*

*Annotations* *Identifier* *Arguments*<sub>opt</sub> *ClassBody*<sub>opt</sub>

*Arguments:*

( *ArgumentList*<sub>opt</sub> )

*EnumBodyDeclarations:*

; *ClassBodyDeclarations*<sub>opt</sub>

枚举常量前面可以放置注释（9.7 节）修饰符。如果枚举常量上的注释 `a` 对应于一个

注释类型  $T$ ，并且  $T$  具有（元）注释  $m$ ，它对应于 `annotation.Target`，那么  $m$  必须具有一个其值为 `annotation.ElementType.FIELD` 的元素，否则就会发生编译时错误。

枚举常量可以后接参数，正如本节后面所描述的，在初始化类期间，当创建枚举常量时，将会把枚举常量后面的参数传递给枚举类型的构造函数。要调用的构造函数是使用普通重载规则（15.12.2 节）选择的。如果遗漏了参数，则会假定一个空参数列表。如果枚举类型没有构造函数声明，则会提供无参的默认构造函数（它匹配隐式的空参数列表）。这个默认构造函数是 `private` 类型。

枚举常量的可选类体隐式定义了一个匿名类声明（15.9.5 节），它扩展了直接封闭的枚举类型。该类体是由匿名类的常用规则控制的：特别是，它不能包含任何构造函数。

#### 讨论

对于这些类体中声明的实例方法，仅当它们重写了封闭枚举类型中的可访问方法时，才可以在封闭枚举类型外部调用它们。

枚举类型（8.9 节）绝对禁止声明为 `abstract`；如果这样声明，将导致一个编译时错误。如果枚举类型  $E$  具有一个 `abstract` 方法  $m$  作为成员，那么除非  $E$  具有一个或多个枚举常量，并且  $E$  的所有枚举常量都具有提供了  $m$  的具体实现的类体，否则就会发生编译时错误。如果枚举常量的类体声明了一个 `abstract` 方法，则会发生编译时错误。

枚举类型隐式地是 `final` 类型，除非它包含至少一个具有类体的枚举常量。无论如何，显式地把枚举类型声明为 `final` 会导致编译时错误。

嵌套的枚举类型隐式地是 `static` 类型。允许显式地把嵌套的枚举类型声明为 `static`。

#### 讨论

这暗示，不可能定义本地（14.3 节）枚举，或者在内部类（8.1.3 节）中定义枚举。

枚举声明内的任何构造函数或成员声明都适用于枚举类型，这就好像它们存在于普通类声明的类体中一样，除非显式指出了另外的情况。

名为  $E$  的枚举类型的直接超类是 `Enum<E>`。除了从 `Enum<E>` 继承的成员之外，对于每个声明的名为  $n$  的枚举常量，枚举类型都有一个隐式声明的名为  $n$ 、类型为  $E$  的公共 `static final` 字段。在枚举类型中显式地声明任何静态字段之前，这些字段将被视为是与对应的枚举常量相同的顺序声明的。每个这样的字段都会被初始化成与之对应的枚举常量。每个这样的字段还会被视作通过与对应的枚举常量相同的注释来为其加注释。当初始化对应的字段时，就称创建了枚举常量。

如果枚举声明一个终结器（`finalizer`），则会发生编译时错误。枚举的实例可能永远不会被销毁。

此外，如果  $E$  是枚举类型的名称，则该类型具有下列隐式声明的静态方法：

```
/**  
 * Returns an array containing the constants of this enum
```

```

* type, in the order they're declared. This method may be
* used to iterate over the constants as follows:
*
*     for(E c : E.values())
*         System.out.println(c);
*
* @return an array containing the constants of this enum
* type, in the order they're declared
*/
public static E[] values();
/**
* Returns the enum constant of this type with the specified
* name.
* The string must match exactly an identifier used to declare
* an enum constant in this type. (Extraneous whitespace
* characters are not permitted.)
*
* @return the enum constant with the specified name
* @throws IllegalArgumentException if this enum type has no
* constant with the specified name
*/
public static E valueOf(String name);

```

### 讨论

枚举类型声明中包含的字段不能与枚举常量相冲突，并且包含的方法不能与自动生成的方法（`values()`和`valueOf(String)`）相冲突，也不能重写 `Enum` 中的 `final` 方法（`equals(Object)`、`hashCode()`、`clone()`、`compareTo(Object)`、`name()`、`ordinal()`和`getDeclaringClass()`）。

如果从枚举类型的构造函数、实例初始化语句块或实例变量初始化语句表达式引用非编译时常量（15.28 节）的枚举类型的静态字段，则会发生编译时错误。如果枚举常量 `e` 的构造函数、实例初始化语句块或实例变量初始化表达式引用其自身，或者引用声明为 `e` 的右边相同类型的枚举常量，则会发生编译时错误。

### 讨论

如果没有这条规则，则由于枚举类型固有的初始化循环性，而使得明显合理的代码在运行时失败（循环性存在于具有“自类型化”静态字段的任何类中）。下面是将会失败的代码类型的示例：

```

enum Color {
    RED, GREEN, BLUE;
    static final Map<String,Color> colorMap =
        new HashMap<String,Color>();
    Color() {

```

```

        colorMap.put(toString(), this);
    }
}

```

这个枚举类型的静态初始化将会抛出一个 `NullPointerException`，这是由于当运行针对枚举常量的构造函数时，静态变量 `colorMap` 未被初始化。上述限制确保这种代码不会通过编译。

注意，该示例可以轻松地被重构成正确地工作：

```

enum Color {
    RED, GREEN, BLUE;
    static final Map<String,Color> colorMap =
        new HashMap<String,Color>();

    static {
        for (Color c : Color.values())
            colorMap.put(c.toString(), c);
    }
}

```

重构的版本显然是正确的，因为静态初始化从代码顶部发生到代码底部。

### 讨论

下面是一个具有嵌套枚举声明的程序，它使用增强的 `for` 循环来迭代枚举中的常量：

```

public class Example1 {
    public enum Season { WINTER, SPRING, SUMMER, FALL }

    public static void main(String[] args) {
        for (Season s : Season.values())
            System.out.println(s);
    }
}

```

运行这个程序，产生如下输出：

```

WINTER
SPRING
SUMMER
FALL

```

下面这个程序阐释了使用 `EnumSet` 来处理子范围：

```

import java.util.*;

public class Example2 {
    enum Day { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY }

    public static void main(String[] args) {
        System.out.print("Weekdays: ");
        for (Day d : EnumSet.range(Day.MONDAY, Day.FRIDAY))
            System.out.print(d + " ");
    }
}

```

```
System.out.println();
```

运行这个程序，产生如下输出：

```
Weekdays: MONDAY TUESDAY WEDNESDAY THURSDAY FRIDAY
```

EnumSet 包含一系列丰富的静态工厂 (static factory)，因此可以把这项技术推广到处理非连续的子集以及子范围。乍看上去，为单个迭代生成一个 EnumSet 显得有些浪费，但它们是如此廉价，以至于这成为迭代子范围所建议的技术。EnumSet 在内部是用单个长整型表示的，假定枚举类型具有 64 个或更少的元素。

下面是一个枚举类型的稍微复杂一些的枚举声明，该枚举类型带有一个显式实例字段以及一个用于该字段的访问器。每个成员在该字段中都有一个不同的值，并且这些值是通过构造函数传入的。在这个示例中，字段表示美国硬币的价值（单位为美分）。但是注意，对可能传递给枚举构造函数的参数的类型或数量没有任何限制。

```
public enum Coin {  
    PENNY(1), NICKEL(5), DIME(10), QUARTER(25);  
    Coin(int value) { this.value = value; }  
    private final int value;  
    public int value() { return value; }  
}
```

switch 语句对于模拟从枚举类型外部添加方法给该类型方法很有用。这个示例“添加了”一个 color 方法给 Coin 类型，并且输出一份表格，其中列出了硬币、它们的价值以及它们的颜色。

```
public class CoinTest {  
    public static void main(String[] args) {  
        for (Coin c : Coin.values())  
            System.out.println(c + ": " + c.value() + "¢ " + color(c));  
    }  
    private enum CoinColor { COPPER, NICKEL, SILVER }  
    private static CoinColor color(Coin c) {  
        switch(c) {  
            case PENNY:  
                return CoinColor.COPPER;  
            case NICKEL:  
                return CoinColor.NICKEL;  
            case DIME: case QUARTER:  
                return CoinColor.SILVER;  
            default:  
                throw new AssertionError("Unknown coin: " + c);  
        }  
    }  
}
```

运行这个程序，产生如下输出：

PENNY:	1¢	COPPER
NICKEL:	5¢	NICKEL
DIME:	10¢	SILVER
QUARTER:	25¢	SILVER

在下面的示例中，在两个简单的枚举类型之上构建了一个玩纸牌的类。注意，在缺少枚举工具的情况下，每个枚举类型将与整个示例一样长：

```
import java.util.*;
public class Card implements Comparable<Card>, java.io.Serializable
{
    public enum Rank { DEUCE, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN, JACK,
    QUEEN, KING, ACE }
    public enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }
    private final Rank rank;
    private final Suit suit;

    private Card(Rank rank, Suit suit) {
        if (rank == null || suit == null)
            throw new NullPointerException(rank + ", " + suit);
        this.rank = rank;
        this.suit = suit;
    }

    public Rank rank() { return rank; }
    public Suit suit() { return suit; }
    public String toString() { return rank + " of " + suit; }
    // Primary sort on suit, secondary sort on rank
    public int compareTo(Card c) {
        int suitCompare = suit.compareTo(c.suit);
        return (suitCompare != 0 ? suitCompare : rank.compareTo(c.rank));
    }

    private static final List<Card> prototypeDeck = new ArrayList<Card>(52);
    static {
        for (Suit suit : Suit.values())
            for (Rank rank : Rank.values())
                prototypeDeck.add(new Card(rank, suit));
    }
    // Returns a new deck
    public static List<Card> newDeck() {
        return new ArrayList<Card>(prototypeDeck);
    }
}
```

下面是一个练习 Card 类的小程序。它从命令行获取两个整型参数，表示需要发牌的人手数量和每只手中的纸牌数量：

```
import java.util.*;
class Deal {
    public static void main(String args[]) {
        int numHands = Integer.parseInt(args[0]);
        int cardsPerHand = Integer.parseInt(args[1]);
        List<Card> deck = Card.newDeck();
        Collections.shuffle(deck);
```



```

        for (int i=0; i < numHands; i++)
            System.out.println(dealHand(deck, cardsPerHand));
    }
    /**
     * Returns a new ArrayList consisting of the last n elements of
     * deck, which are removed from deck. The returned list is
     * sorted using the elements' natural ordering.
     */
    public static <E extends Comparable<E>> ArrayList<E>
        dealHand(List<E> deck, int n) {
        int deckSize = deck.size();
        List<E> handView = deck.subList(deckSize - n, deckSize);
        ArrayList<E> hand = new ArrayList<E>(handView);
        handView.clear();
        Collections.sort(hand);
        return hand;
    }
}

```

运行这个程序，产生如下结果：

```

java Deal 4 5
[FOUR of SPADES, NINE of CLUBS, NINE of SPADES, QUEEN of SPADES,
KING of SPADES]
[THREE of DIAMONDS, FIVE of HEARTS, SIX of SPADES, SEVEN of DIAMONDS,
KING of DIAMONDS]
[FOUR of DIAMONDS, FIVE of SPADES, JACK of CLUBS, ACE of DIAMONDS, ACE of HEARTS]
[THREE of HEARTS, FIVE of DIAMONDS, TEN of HEARTS, JACK of HEARTS,
QUEEN of HEARTS]

```

下面的示例展示了使用特定于常量的类体将行为附加到常量上（预期很少有人需要使用该程序）：

```

import java.util.*;

public enum Operation {
    PLUS {
        double eval(double x, double y) { return x + y; }
    },
    MINUS {
        double eval(double x, double y) { return x - y; }
    },
    TIMES {
        double eval(double x, double y) { return x * y; }
    },
    DIVIDED_BY {
        double eval(double x, double y) { return x / y; }
    };
    // Perform the arithmetic operation represented by this constant
    // abstract double eval(double x, double y);
    public static void main(String args[]) {

```

```
double x = Double.parseDouble(args[0]);
double y = Double.parseDouble(args[1]);
for (Operation op : Operation.values())
    System.out.println(x+"*"+op+"*"+y+"="+ op.eval(x, y));
}
```

运行这个程序，产生如下输出：

```
java Operation 2.0 4.0
2.0 PLUS 4.0 = 6.0
2.0 MINUS 4.0 = -2.0
2.0 TIMES 4.0 = 8.0
2.0 DIVIDED_BY 4.0 = 0.5
```

上述模式适合于能力适中的程序员。无可否认，它有一点技巧性，但是它比在基类型（运算）中使用 case 语句要安全得多，因为该模式排除了忘记为新常量添加行为这种可能性（如果这发生的话，将获得一个编译时错误）。

鞠躬、鞠躬，尔等中下阶层！  
鞠躬、鞠躬，尔等商人，鞠躬、尔等民众！  
吹响喇叭，敲响铜锣！  
——W.S.Gilbert, 《Iolanthe》

## 接口

我的苹果树永远不会越过栅栏吃掉松树的球果，我告诉他。  
他只说了一句：“好栅栏产生好邻居。”  
——罗伯特·弗罗斯特，《补墙》（1914）

接口声明引入了一种新的引用类型，其成员有：类、接口、常量和抽象方法。这种类型没有实现，但是其他不相关的类可以通过为其抽象方法提供实现来实现它。

嵌套接口是其声明出现在另一个类或接口的主体内的任何接口。顶级接口是并非嵌套接口的接口。

我们把接口分为两类——普通接口和注释类型。

本章讨论了所有接口——普通接口和注释类型（9.6 节）、顶级接口（7.6 节）和嵌套接口（8.5 节、9.5 节）的公共语义。在下面专门针对这些构造的几节中讨论了特殊类型的接口特有的详细信息。

程序可以使用接口，使得相关的类不必共享公共抽象超类，或者添加方法到 `Object` 中。

接口可以被声明成一个或多个其他接口的直接扩展，这意味着除了可能隐藏的任何成员类型和常量之外，它隐式指定了它所扩展的接口的所有成员类型、抽象方法和常量。

类可以被声明成直接实现了一个或多个接口，这意味着类的任何实例都实现了通过一个或多个接口指定的所有抽象方法。类必须实现其直接超类和直接超接口所实现的所有接口。这种（多重）接口继承性允许对象支持（多种）公共行为，而无需共享任何实现。

其声明类型为接口类型的变量可以把其值作为一个指向类的任何实例的引用，该类实现了指定的接口。类碰巧实现了接口的所有抽象方法是不够的：类或其超类之一实际上必须被声明成实现接口，否则，就不认为类实现了接口。

## 9.1 接口声明

接口声明指定了一种新命名的引用类型。接口声明有两种——普通接口声明和注释类型声明：

*InterfaceDeclaration:*

*NormalInterfaceDeclaration*

*AnnotationTypeDeclaration*

第 9.6 节中进一步描述了注释类型。

*NormalInterfaceDeclaration:*

```
InterfaceModifiersopt interface Identifier TypeParametersopt
                               ExtendsInterfacesopt InterfaceBody
```

接口声明中的 *Identifier* 指定了接口的名称。如果一个接口具有与其封闭类或接口相同的简单名称，则会发生编译时错误。

### 9.1.1 接口修饰符

接口声明可以包括接口修饰符：

*InterfaceModifiers:*

*InterfaceModifier*

*InterfaceModifiers* *InterfaceModifier*

*InterfaceModifier: one of*

```
Annotation public protected private
abstract static strictfp
```

第 6.6 节中讨论了访问修饰符 `public`。并非所有的修饰符都适用于各种类型的接口声明。访问修饰符 `protected` 和 `private` 只适合于第 8.5.1 节中讨论的直接封闭类声明 (8.5 节) 内的成员接口。访问修饰符 `static` 只适合于成员接口 (8.5 节、9.5 节)。如果同一个访问修饰符在接口声明中出现了不止一次，则会发生编译时错误。如果接口声明上的注释 *a* 对应于一个注释类型 *T*，并且 *T* 具有一个 (元) 注释 *m*，它对应于 `annotation.Target`，那么 *m* 必须具有一个元素，其值是 `annotation.ElementType.TYPE`，否则就会发生编译时错误。第 9.7 节中进一步描述了注释修饰符。

#### 9.1.1.1 abstract 接口

每个接口隐式地都是 `abstract` 接口。这个修饰符是过时的，不应该在新程序中使用它。

#### 9.1.1.2 strictfp 接口

`strictfp` 修饰符的作用是使接口声明内的所有 `float` 或 `double` 表达式显式地都是精确浮点的 (15.4 节)。

这暗示接口中声明的所有嵌套类型隐式地都是 `strictfp`。

### 9.1.2 泛型接口和类型参数

如果一个接口声明了一个或多个类型变量 (4.4 节)，则该接口就是泛型。这些类型变

量被称为接口的类型参数。类型参数部分接在接口名称后面，并通过尖括号进行定界。它定义了一个或多个用作参数的类型变量。泛型接口声明定义了一组类型，每个类型用于类型参数部分的每个可能的调用。所有参数化类型在运行时共享相同的接口。

接口的类型参数的作用域是接口的整个声明，包括类型参数部分本身。因此，类型参数可以作为它们自身界限的一部分出现，或者作为同一部分中声明的其他类型参数的界限出现。

在接口  $I$  的字段或类型成员声明中的任何位置引用接口  $I$  的类型参数，将会导致编译时错误。

### 9.1.3 超接口和子接口

如果提供了一个 `extends` 子句，那么被声明的接口就会扩展其他每个命名接口，从而继承其他每个命名接口的成员类型、方法和常量。这些其他的命名接口是被声明的接口的直接超接口。实现被声明接口的任何类还会被视作实现了这个接口扩展的所有接口。

*ExtendsInterfaces:*

`extends InterfaceType`

*ExtendsInterfaces, InterfaceType*

下面重复了第 4.3 节中的内容，以使这里的介绍更清楚：

*InterfaceType:*

*TypeDeclSpecifier TypeArguments<sub>opt</sub>*

给定  $I\langle F_1, \dots, F_n \rangle$ （其中  $n \geq 0$ ）的（可能是泛型）接口声明，接口类型（4.5 节） $I\langle F_1, \dots, F_n \rangle$  的直接超接口是  $I$  的声明的 `extends` 子句（如果 `extends` 子句存在的话）中给定的类型。

设  $I\langle F_1, \dots, F_n \rangle$ （其中  $n > 0$ ）是泛型接口声明。参数化接口类型  $I\langle T_1, \dots, T_n \rangle$ （其中  $T_i$ （ $1 \leq i \leq n$ ）是一个类型）的直接超接口是所有类型  $J\langle U_1 \text{ theta}, \dots, U_k \text{ theta} \rangle$ ，其中  $J\langle U_1, \dots, U_k \rangle$  是  $I\langle F_1, \dots, F_n \rangle$  的直接超接口， $\text{theta}$  是代换  $[F_1 := T_1, \dots, F_n := T_n]$ 。

接口声明的 `extends` 子句中的每个 *InterfaceType* 必须指定一种可访问的接口类型；否则，就会发生编译时错误。

如果在接口  $I$  的 `extends` 子句中将类型  $T$  称为超接口或者超接口名称内的限定符，则接口  $I$  直接依赖于类型  $T$ 。如果以下任何一个条件成立，则接口  $I$  依赖于引用类型  $T$ ：

- $I$  直接依赖于  $T$ 。
- $I$  直接依赖于类  $C$ ， $C$  依赖（8.1.5 节）于  $T$ 。
- $I$  直接依赖于接口  $J$ ， $J$  依赖于  $T$ （递归地使用这个定义）。

如果一个接口依赖于它自身，则会发生编译时错误。

虽然每个类都是类 `Object` 的扩展，但是所有接口都不是其扩展的单独一个接口。

超接口关系是直接超接口关系的传递闭包（transitive closure）。如果以下条件之一成立，则接口  $K$  是接口  $I$  的超接口：

- $K$  是  $I$  的直接超接口。
- 存在一个接口  $J$ , 其中  $K$  是  $J$  的超接口, 并且  $J$  是  $I$  的超接口, 递归地应用这个定义。只要接口  $K$  是接口  $I$  的超接口, 则称接口  $I$  是接口  $K$  的子接口。

### 9.1.4 接口体和成员声明

接口的体可以声明接口的成员:

*InterfaceBody:*

{ *InterfaceMemberDeclarations*<sub>opt</sub> }

*InterfaceMemberDeclarations:*

*InterfaceMemberDeclaration*

*InterfaceMemberDeclarations* *InterfaceMemberDeclaration*

*InterfaceMemberDeclaration:*

*ConstantDeclaration*

*AbstractMethodDeclaration*

*ClassDeclaration*

*InterfaceDeclaration*

;

在接口类型  $I$  中声明或被其继承的成员  $m$  声明的作用域是  $I$  的整个主体, 包括任何嵌套的类型声明。

### 9.1.5 访问接口成员名称

所有的接口成员隐式地都是 `public`。依据第 6.6 节中的规则, 如果接口被声明为 `public` 或 `protected`, 则在声明接口的包外部可以访问接口成员。

## 9.2 接口成员

接口的成员有:

- 在接口中声明的那些成员。
- 从直接超接口继承的那些成员。
- 如果一个接口没有直接超接口, 那么这个接口就会隐式声明一个具有签名  $s$ 、返回类型  $r$  和 `throws` 子句的公共抽象成员方法  $m$ , 它对应于 `Object` 中声明的具有签名  $s$ 、返回类型  $r$  和 `throws` 子句的每个公共实例成员方法  $m$ , 除非该接口显式声明了一个具有相同签名、相同返回类型和兼容的 `throws` 子句的方法。如果该接口显式声明了这样一个方法  $m$ , 其中  $m$  在 `Object` 中被声明为 `final`, 则会发生编译时错误。



如果接口声明了一个方法，该方法的签名与 `Object` 的公共方法是重写等价的，但是具有不同的返回类型或不兼容的 `throws` 子句，则会发生编译时错误（8.4.2 节）。

接口会从其扩展的接口继承那些接口的所有成员，除了它隐藏的字、类和接口以及它重写的方法之外。

## 9.3 字段（常量）声明

万变不离其宗。

——埃皮克提图（公元前一世纪的希腊哲学家）

*ConstantDeclaration:*

*ConstantModifiers*<sub>opt</sub> *Type* *VariableDeclarators* ;

*ConstantModifiers:*

*ConstantModifier*

*ConstantModifier ConstantModifiers*

*ConstantModifier: one of*

*Annotation* `public` `static` `final`

接口体中的每个字段声明隐式地都是 `public`、`static` 和 `final`。允许为这些字段冗余地指定任何或所有这些修饰符。

如果字段声明上的注释 *a* 对应于一个注释类型 *T*，并且 *T* 具有一个（元）注释 *m*，它对应于 `annotation.Target`，那么 *m* 必须具有一个元素，其值是 `annotation.ElementType.FIELD`，否则就会发生编译时错误。第 9.7 节中进一步描述了注释修饰符。

如果接口用某个名称声明了一个字段，那么就称该字段的声明隐藏了那个接口的超接口中具有相同名称字段的任何和所有可访问的声明。

如果接口声明的主体中声明了两个同名字段，则会发生编译时错误。

接口有可能继承多个具有相同名称的字段（8.3.3 节）。这样的情形本质上不会引发编译时错误。但是，在接口体内试图通过其简单名称引用任何这样的字段，都会导致编译时错误，这是因为这样的引用是有歧义的。

可以通过多条途径从一个接口继承相同的字段声明。在这种情形下，将字段视为只会被继承一次，并且可以通过其简单名称无歧义地引用它。

### 9.3.1 接口中的字段初始化

接口体中的每个字段都必须具有一个初始化表达式，它不必是一个常量表达式。当初始化接口时（12.4 节），会恰好执行一次变量初始化语句的求值和赋值。

如果接口字段的初始化表达式包含一个引用，该引用通过简单名称指向相同的字段，或者指向其声明出现在相同接口的代码后面的另一个字段，则会发生编译时错误。

因此：

```
interface Test {  
    float f = j;  
    int j = 1;  
    int k = k+1;  
}
```

会引起两个编译时错误，这是由于在声明 `j` 之前，在 `f` 的初始化中引用了 `j`；并且由于 `k` 的初始化引用了 `k` 本身。

这里的一个微妙之处是：在运行时，通过编译时常量值进行初始化的字段会最先被初始化。这也适用于类中的 `static final` 字段（8.3.2.1 节）。特别地，这意味着这些字段永远不会被观察到具有其默认初始值（4.12.5 节），甚至通过迂回的程序也是如此。更多讨论请参见第 12.4.2 节和第 13.4.9 节。

如果接口字段的初始化表达式中出现了关键字 `this`（15.8.3 节）或关键字 `super`（15.11.2 节、15.12 节），那么除非它们出现在匿名类（15.9.5 节）的类体中，否则就会发生编译时错误。

### 9.3.2 字段声明的示例

下列示例阐释了关于字段声明的一些（可能是微妙的）要点。

#### 9.3.2.1 有歧义的继承字段

例如，如果由于某个接口的两个直接超接口用同一个名称声明了某个字段，而使得该接口继承了两个同名字段，则会产生单个有歧义的成员。无论怎样使用这个有歧义的成员，都会导致一个编译时错误。

因此，在下面的示例中：

```
interface BaseColors {  
    int RED = 1, GREEN = 2, BLUE = 4;  
}  
interface RainbowColors extends BaseColors {  
    int YELLOW = 3, ORANGE = 5, INDIGO = 6, VIOLET = 7;  
}  
interface PrintColors extends BaseColors {  
    int YELLOW = 8, CYAN = 16, MAGENTA = 32;  
}  
interface LotsOfColors extends RainbowColors, PrintColors {  
    int FUCHSIA = 17, VERMILION = 43, CHARTREUSE = RED+90;  
}
```

接口 `LotsOfColors` 继承两个名为 `YELLOW` 的字段。只要该接口中不包含任何通过简单名称指向字段 `YELLOW` 的引用，这就是合理的（这种引用可能出现在字段的变量初始化语句内）。

即使接口 `PrintColors` 将数值 3 (而不是数值 8) 赋予 `YELLOW`, 指向接口 `LotsOfColors` 内字段 `YELLOW` 的引用仍将会被视作是有歧义的。

### 9.3.2.2 多重继承的字段

例如, 如果由于某个接口及其直接超接口之一扩展了声明某个字段的接口, 而使得从同一个接口对单个字段继承了多次, 那么只会产生单个成员。这种情形本质上不会引发编译时错误。

在前一节的那个示例中, 接口 `LotsOfColors` 通过接口 `RainbowColors` 和接口 `PrintColors` 以多种方式继承了字段 `RED`、`GREEN` 和 `BLUE`, 但是指向接口 `LotsOfColors` 中的字段 `RED` 的引用不会被视作是有歧义的, 这是由于只涉及了字段 `RED` 的一个实际的声明。

## 9.4 抽象方法声明

*AbstractMethodDeclaration:*

*AbstractMethodModifiers<sub>opt</sub> TypeParameters<sub>opt</sub> ResultType  
MethodDeclarator Throws<sub>opt</sub>;*

*AbstractMethodModifiers:*

*AbstractMethodModifier*

*AbstractMethodModifiers AbstractMethodModifier*

*AbstractMethodModifier: one of*

*Annotation public abstract*

第 6.6 节中讨论了访问修饰符 `public`。如果同一个修饰符在抽象方法声明中出现了不止一次, 则会发生编译时错误。

接口体中的每个方法声明隐式地都是 `abstract`, 因此, 总是用分号 (而不是代码块) 表示接口体。

接口体中的每个方法声明隐式地都是 `public`。

为了保持与 Java 平台旧版本的兼容性, 作为一种编程风格, 允许 (但不鼓励) 为接口中声明的方法冗余地指定 `abstract` 修饰符。

作为一种编程风格, 允许 (但强烈不鼓励) 为接口方法冗余地指定 `public` 修饰符。

注意, 绝对禁止把接口中声明的方法声明为 `static`, 否则就会发生编译时错误, 这是由于 `static` 方法不能是 `abstract`。

注意, 绝对禁止把接口中声明的方法声明为 `strictfp`、`native` 或 `synchronized`, 否则就会发生编译时错误, 这是由于这些关键字描述的是实现的属性, 而不是接口的属性。但是, 在接口中声明的方法可以通过在实现该接口的类中声明为 `strictfp`、`native` 或 `synchronized` 的方法来实现。

如果方法声明上的注释 `a` 对应于一个注释类型 `T`, 并且 `T` 具有一个 (元) 注释 `m`, 它对应于 `annotation.Target`, 那么 `m` 必须具有一个元素, 其值是 `annotation`。

ElementType.METHOD, 否则就会发生编译时错误。第 9.7 节中进一步描述了注释修饰符。

如果接口显式或隐式地声明两个具有重写等价签名 (8.4.2 节) 的方法, 则会发生编译时错误。但是, 一个接口可以继承多个具有这种签名的方法 (9.4.1 节)。

注意, 绝对禁止把接口中声明的方法声明为 final, 否则就会发生编译时错误。但是, 在接口中声明的方法可以通过在实现该接口的类中声明为 final 的方法来实现。

接口中的方法可以是泛型。针对接口中的泛型方法的形式类型参数的规则与针对类中的泛型方法的规则一样 (8.4.4 节)。

### 9.4.1 继承和重写

当且仅当下列两个条件都成立时, 接口  $I$  中声明的实例方法  $m_1$  才会重写接口  $J$  中声明的另一个实例方法  $m_2$ :

- (1)  $I$  是  $J$  的子接口。
- (2)  $m_1$  的签名是  $m_2$  的签名的子签名 (8.4.2 节)。

如果返回类型为  $R_1$  的方法声明  $d_1$  重写或隐藏了另一个返回类型为  $R_2$  的方法  $d_2$  的声明, 那么  $d_1$  必须是  $d_2$  的可替代返回类型 (8.4.5 节), 否则就会发生编译时错误。此外, 如果  $R_1$  不是  $R_2$  的子类型, 则必须发出未经检查的警告。

此外, 方法声明绝对禁止具有与它所重写的任何方法的 throws 子句相冲突 (8.4.6 节) 的 throws 子句; 否则, 就会发生编译时错误。

如果类型声明  $T$  具有一个成员方法  $m_1$ , 并且存在  $T$  或其超类型中声明的方法  $m_2$ , 使得以下所有条件均成立, 则会发生编译时错误:

- $m_1$  和  $m_2$  具有相同的名称。
- 可以从  $T$  访问  $m_2$ 。
- $m_1$  的签名不是  $m_2$  的签名的子签名 (8.4.2 节)。
- $m_1$  或方法  $m_1$  的某个重写版本 (直接或间接) 具有与  $m_2$  或方法  $m_2$  的某个重写版本 (直接或间接) 相同的擦除。

方法是在逐个签名的基础上进行重写的。例如, 如果接口声明了两个具有相同名称的 public 方法, 并且一个子接口重写了其中一个方法, 则该子接口仍会继承另一个方法。

一个接口会从其直接超接口继承这些超接口的所有未被该接口中的声明重写的方法。

一个接口有可能继承多个具有重写等价签名 (8.4.2 节) 的方法。这种情形本质上不会引发编译时错误。该接口将被视作继承了所有的方法。但是, 被继承的方法之一对于任何其他被继承的方法必须是返回类型可替代的; 否则, 就会发生编译时错误 (在这种情况下, throws 子句不会引发错误)。

可以通过多条途径从一个接口继承相同的方法声明。这个事实不会引起难以解决的情况, 并且其自身永远不会导致编译时错误。

### 9.4.2 重载

如果接口的两个方法 (无论它们是在同一个接口中声明的, 或者都是被接口继承的,

抑或一个是声明的另一个是继承的)具有相同的名称,但是其签名不是重写等价的(8.4.2节),那么就称方法名称是被重载的。这个事实不会引起难以解决的情况,并且其自身永远不会导致编译时错误。在具有相同名称但并非重写等价的不同签名的两个方法的返回类型之间或者 throws 子句之间不需要有任何关系。

### 9.4.3 抽象方法声明的示例

下面的示例阐释了有关抽象方法声明的一些(可能是微妙的)要点。

#### 9.4.3.1 示例: 重写

接口中声明的方法都是 abstract, 因而不含任何实现。除了确认方法签名之外, 关于能够被重写方法声明完成的一切是精炼返回类型, 或者限制可以被方法的实现抛出的异常。下面是第 8.4.3.1 节中所示示例的一个变体:

```
class BufferEmpty extends Exception {
    BufferEmpty() { super(); }
    BufferEmpty(String s) { super(s); }
}
class BufferException extends Exception {
    BufferException() { super(); }
    BufferException(String s) { super(s); }
}
public interface Buffer {
    char get() throws BufferEmpty, BufferException;
}
public interface InfiniteBuffer extends Buffer {
    char get() throws BufferException; // override
}
```

#### 9.4.3.2 示例: 重载

在下面的示例代码中:

```
interface PointInterface {
    void move(int dx, int dy);
}
interface RealPointInterface extends PointInterface {
    void move(float dx, float dy);
    void move(double dx, double dy);
}
```

在接口 RealPointInterface 中用 3 个不同的签名重载方法名称 move, 其中两个签名是声明的, 另一个是继承的。实现接口 RealPointInterface 的任何非 abstract 类都必须提供所有这 3 个方法签名的实现。

## 9.5 成员类型声明

接口可以包含成员类型声明(8.5 节)。接口中的成员类型声明隐式地都是 static 和 public。

如果用简单名称 *C* 声明的成员类型被直接封闭在具有完全限定名称 *N* 的接口的声明内, 则该成员类型具有完全限定名称 *N.C*。

如果接口用某个名称声明了一个成员类型, 那么就称该成员类型的声明隐藏了那个接口的超接口中具有相同名称的成员类型任何和所有可访问的声明。

接口从其直接超接口继承了这些超接口的所有非私有成员类型, 这些超接口对于接口中的代码都是可访问的, 并且不会被接口中的声明隐藏。

接口可以继承两个或多个具有相同名称的类型声明。如果试图通过其简单名称引用任何有歧义地继承的类或接口, 则会发生编译时错误。如果通过多条途径从一个接口继承相同的类型声明, 则会将类或接口视作只被继承一次; 可以通过其简单名称无歧义地引用它。

## 9.6 注释类型

注释类型声明是一种特殊的接口声明。为了把注释类型声明与普通接口声明区分开, 可以在关键字 `interface` 前面放置一个 `at` 符号 (`@`)。



注意, `at` 符号 (`@`) 和关键字 `interface` 是两个不同的标记; 从技术上讲, 可以用空白把它们隔开, 但强烈反对这种编程风格。

*AnnotationTypeDeclaration:*

*InterfaceModifiers<sub>opt</sub> @ interface Identifier AnnotationTypeBody*

*AnnotationTypeBody:*

*[ AnnotationTypeElementDeclarations<sub>opt</sub> ]*

*AnnotationTypeElementDeclarations:*

*AnnotationTypeElementDeclaration*

*AnnotationTypeElementDeclarations AnnotationTypeElementDeclaration*

*AnnotationTypeElementDeclaration:*

*AbstractMethodModifiers<sub>opt</sub> Type Identifier ( ) DefaultValue<sub>opt</sub> ;*

*ConstantDeclaration*

*ClassDeclaration*

*InterfaceDeclaration*

*EnumDeclaration*

*AnnotationTypeDeclaration*

*;*

*DefaultValue:*

*default ElementValue*



**讨论**

下列限制通过其与环境无关的优点而强加到注释类型声明上：

- 注释类型声明不能是泛型。
- 不允许使用 `extends` 子句（注释类型隐式扩展 `annotation.Annotation`）。
- 方法不能有任何参数。
- 方法不能有任何类型参数。
- 方法声明不能有 `throws` 子句。

从此以后，除非被明确修改，否则，适用于普通接口声明的所有规则也适用于注释类型声明。

**讨论**

例如，注释类型作为普通类和接口类型共享相同的命名空间。

在接口声明合法的任何位置，注释类型声明也是合法，并且具有相同的作用域和可访问性。

注释类型声明中的 *Identifier* 指定了注释类型的名称。如果注释类型具有与其任何封闭类或接口相同的简单名称，则会发生编译时错误。

如果注释类型声明上的注释 *a* 对应于一个注释类型 *T*，并且 *T* 具有一个（元）注释 *m*，它对应于 `annotation.Target`，那么 *m* 必须具有一个元素，其值是 `annotation.ElementType.ANNOTATION_TYPE`，否则就会发生编译时错误。

**讨论**

根据约定，除了注释之外，不应该存在 *AbstractMethodModifiers*。

注释类型的直接超接口总是 `annotation.Annotation`。

**讨论**

注释类型不能显式声明超类或超接口，这一事实的后果是：注释类型自身的子类或子接口永远不会是一个注释类型。类似地，`annotation.Annotation` 自身也不是一个注释类型。

如果注释类型中声明的方法的返回类型是除以下类型以外的任何类型：基本类型之一、`String`、`Class` 和 `Class` 的任何调用、枚举类型（8.9 节）、注释类型或上述类型之一的数组（第 10 章），则会发生编译时错误。如果注释类型中声明的方法的签名与类 `Object` 或接口 `annotation.Annotation` 中声明的任何 `public` 或 `protected` 方法的签名是重写等价的，则会发生编译时错误。

---

**讨论**

注意，这不与泛型方法上的禁令相冲突，因为通配符消除了对显式类型参数的需要。

---

注释类型声明中的每个方法声明定义了注释类型的一个元素。注释类型可以具有零个或多个元素。除了通过其显式声明的方法定义的那些元素外，注释类型不具有其他任何元素。

---

因此，注释类型声明从 `annotation.Annotation` 继承了多个成员，包括隐式声明的与 `Object` 中的实例方法对应的方法，然而这些方法并没有定义注释类型的元素，并且在注释中使用它们是非法的。

如果没有这条规则，我们就不能确保注释中的元素是类型可表示的，或者针对它们的访问方法是可用的。

---

如果注释类型  $T$ （直接或间接地）包含一个类型  $T$  的元素，则会发生编译时错误。

---

例如，下面的示例是非法的：

```
// Illegal self-reference!!
@interface SelfRef {
    SelfRef value();
}
```

下面的示例也是非法的：

```
// Illegal circularity!!
@interface Ping {
    Pong value();
}
@interface Pong {
    Ping value();
}
```

另请注意，本规范把类型为嵌套数组的元素排除在外。例如，下面的注释类型声明是非法的：

```
// Illegal nested array!!
@interface Verboten {
    String[][] value();
}
```

---

注释类型元素可以具有为其指定的默认值。这是通过在其（空）参数列表后面接上关键字 `default` 和元素的默认值来完成的。

默认值是在阅读注释时动态应用的：默认值不会编译进注释中。因此，在执行更改前，更改默认值将会影响被编译的注释，甚至在类中也是如此（假定这些注释没有为默认元素提供一个显式值）。

*ElementValue* 用于指定一个默认值。如果元素的类型与指定的默认值不匹配（9.7 节），则会发生编译时错误。*ElementValue* 总是精确浮点的（15.4 节）。

#### 讨论

下面的注释类型声明定义了一个具有多个元素的注释类型：

```
// Normal annotation type declaration with several elements
/**
 * Describes the "request-for-enhancement" (RFE)
 * that led to the presence of
 * the annotated API element.
 */
public @interface RequestForEnhancement {
    int id();           // Unique ID number associated with RFE
    String synopsis();  // Synopsis of RFE
    String engineer();   // Name of engineer who implemented RFE
    String date();       // Date RFE was implemented
}
```

下列注释类型声明定义了一个不含元素的注释类型，称为标记注释类型（marker annotation type）：

```
// Marker annotation type declaration
/**
 * Annotation with this type indicates that the specification of the
 * annotated API element is preliminary and subject to change.
 */
public @interface Preliminary { }
```

根据约定，单元素注释类型中的惟一元素的名称是 value。

#### 讨论

对这个约定的语言学支持是由单元素注释构造（9.7 节）提供的：程序员必须遵守这个约定，以利用这种构造。

#### 讨论

下列注释类型声明中阐释了这个约定：

```
// Single-element annotation type declaration
/**
 * Associates a copyright notice with the annotated API element.
 */
```

```
public @interface Copyright {
    String value();
}
```

下列注释类型声明定义了其惟一元素具有数组类型的单元素注释类型:

```
// Single-element annotation type declaration with array-typed
// element
/**
 * Associates a list of endorsers with the annotated class.
 */
public @interface Endorsers {
    String[] value();
}
```

下面是一个复杂注释类型的示例, 这些注释类型包含一个或多个其类型也是注释类型的元素。

```
// Complex Annotation Type
/**
 * A person's name. This annotation type is not designed to be used
 * directly to annotate program elements, but to define elements
 * of other annotation types.
 */
public @interface Name {
    String first();
    String last();
}
/**
 * Indicates the author of the annotated program element.
 */
public @interface Author {
    Name value();
}
/**
 * Indicates the reviewer of the annotated program element.
 */
public @interface Reviewer {
    Name value();
}
```

下面的注释类型声明为其 4 个元素中的 2 个元素提供了默认值:

```
// Annotation type declaration with defaults on some elements
public @interface RequestForEnhancement {
    int id(); // No default - must be specified in
              // each annotation
    String synopsis(); // No default - must be specified in
                       // each annotation
    String engineer() default "{unassigned}";
}
```

```
String date() default "[unimplemented]";  
}
```

下面的注释类型声明显示了一个 Class 注释，其值受到有界通配符的限制。

```
// Annotation type declaration with bounded wildcard to  
// restrict Class annotation  
// The annotation type declaration below presumes the existence  
// of this interface, which describes a formatter for Java  
// programming language source code  
public interface Formatter { ... }  
// Designates a formatter to pretty-print the annotated class.  
public @interface PrettyPrinter {  
    Class<? extends Formatter> value();  
}
```

注意，注释类型声明的语法允许除方法声明之外的其他元素声明。例如，程序员可以选择声明一个嵌套枚举，用于连接一个注释类型：

```
// Annotation type declaration with nested enum type declaration  
public @interface Quality {  
    enum Level { BAD, INDIFFERENT, GOOD }  
  
    Level value();  
}
```

### 9.6.1 预定义的注释类型

Java 平台的库中预定义了多个注释类型。这些预定义的注释类型中有一些具有特殊的语义。本节中详细说明了这些语义。本节并没有为这里包含的预定义的注释提供完整的规范；它是合适的 API 规范的职责。这里只详细说明了那些在 Java 编译器或虚拟机上需要特殊行为的语义。

#### 9.6.1.1 Target

注释类型 `annotation.Target` 打算用在元注释中，它指定了注释类型所适用的程序元素的种类。`Target` 具有一个 `annotation.ElementType[]` 类型的元素。如果给定的枚举常量在其对应类型是 `annotation.Target` 的注释中出现了不止一次，则会发生编译时错误。有关 `@annotation.Target` 注释的其他作用，参见第 7.4.1 节、8.1.1 节、8.3.1 节、8.4.1 节、8.4.3 节、8.8.3 节、8.9 节、9.1.1 节、9.3 节、9.4 节、9.6 节和 14.4 节。

#### 9.6.1.2 Retention

注释可能只存在于源代码中，或者存在于类或接口的二进制形式中。对于存在于二进制形式中的注释，也许可以也许不能通过 Java 平台的反射库使之在运行时可用。

注释类型 `annotation.Retention` 用于在上述几种可能性之间做出选择。如果注释 `a` 对应于类型 `T`，并且 `T` 具有一个（元）注释 `m`，它对应于 `annotation.Retention`，则：

- 如果 `m` 具有一个其值为 `annotation.RetentionPolicy.SOURCE` 的元素，那么

Java 编译器必须确保 *a* 不会存在于出现 *a* 的类或接口的二进制表示中。

- 如果 *m* 具有一个其值为 `annotation.RetentionPolicy.CLASS` 或 `annotation.RetentionPolicy.RUNTIME` 的元素, 那么 Java 编译器必须确保 *a* 是用出现 *a* 的类或接口的二进制表示来表示的, 除非 *m* 注释了一个局部变量声明。局部变量声明上的注释永远不会保留在二进制表示中。

如果 *T* 不具有一个对应于 `annotation.Retention` 的 (元) 注释 *m*, 那么 Java 编译器必须像它具有这样一个元注释 *m* 并且 *m* 具有一个其值为 `annotation.RetentionPolicy.CLASS` 的元素那样来对待 *T*。

#### 讨论

如果 *m* 具有一个其值为 `annotation.RetentionPolicy.RUNTIME` 的元素, 则 Java 平台的反射库也会在运行时可用。

#### 9.6.1.3 Inherited

注释类型 `annotation.Inherited` 用于指定类 *C* 上的注释, 它对应于被 *C* 的子类继承的给定的注释类型。

#### 9.6.1.4 Override

当程序员打算重写一个方法声明时, 偶尔也可以重载它。

#### 讨论

这个经典示例涉及 `equals` 方法。程序员可以编写如下代码:

```
public boolean equals(Foo that) { ... }
```

当它们打算编写如下代码时:

```
public boolean equals(Object that) { ... }
```

这是非常合法的, 但是类 `Foo` 从 `Object` 继承了 `equals` 实现, 这可能引发一些非常细微的错误。

注释类型 `Override` 支持及早检测到这些问题。如果利用注释 `@Override` 为方法声明加注释, 但是该方法事实上没有重写超类中声明的任何方法, 则会发生编译时错误。

#### 讨论

注意, 如果一个方法重写了超接口中的方法, 但是没有重写超类的方法, 那么使用 `@Override` 将会引发编译时错误。

这条规则的基本原理是: 实现一个接口的具体类必须重写所有接口的方法, 而不考虑 `@Override` 注释, 因此, 把该注释的语义与实现接口的规则相结合将会使人弄混淆。

这条规则的一个副作用是: 永远不可能在接口声明中使用 `@Override` 注释。



### 9.6.1.5 SuppressWarnings

注释类型 SuppressWarnings 支持程序员控制 Java 编译器另外发出的警告。它包含单独一个元素，即一个 String 数组。如果用注释 @SuppressWarnings(value = { $S_1$ , ...,  $S_k$ }) 为程序声明加注释，并且如果警告是由加注释的声明或其任何部分生成的，那么 Java 编译器绝对禁止报告任何通过  $S_1$ , ...,  $S_k$  之一标识的警告。

未经检查的警告是通过字符串 "unchecked" 标识的。

#### 讨论

与以前的旧版本相比，最新的 Java 编译器会发出更多的警告，并且这些“类似绷带布的”警告很有用。随着时间的推移，很可能会添加更多这样的警告。为了鼓励使用它们，当程序员知道某个警告不合适时，应该有某种方式用于在程序的特殊部分禁用该警告。

#### 讨论

编译器供应商应该用文档记录他们与这种注释类型一道支持的警告名称。鼓励他们开展合作，以确保相同的名称能够跨多种编译器正常工作。

### 9.6.1.6 Deprecated

加有 @Deprecated 注释的程序元素是不鼓励程序员使用的元素，这通常是由于它是危险的，或者由于存在更好的替代方案。当使用（通过名称重写、调用或引用）那些不赞成使用的类型、方法、字段或构造函数时，Java 编译器必须产生警告，除非以下条件之一成立：

- 在某个实体中使用它们，并且该实体本身利用注释 @Deprecated 加过注释。
- 声明和使用都是在同一个最外层的类中。
- 在实体中使用它们，利用注释 @SuppressWarnings("deprecation") 给该实体加上注释，以禁止警告。

在局部变量声明或参数声明上使用注释 @Deprecated 将不起作用。

## 9.7 注释

注释是一个修饰符，包含注释类型（9.6 节）的名称以及零个或多个元素-值对，其中每一对都把一个值与注释类型的一个不同的元素相关联。注释的目的是把信息与加有注释的程序元素相关联。

除了那些具有默认值的元素之外，注释必须为相应注释类型的每个元素包含一个元素-值对，否则就会发生编译时错误。注释可以（但是不要求）为具有默认值的元素包含元素-值对。

注释可以用作任何声明中的修饰符，包括：包（7.4 节）、类（第 8 章）、接口、字段（8.3 节、9.3 节）、方法（8.4 节、9.4 节）、参数、构造函数（8.8 节）或局部变量（14.4 节）。

**讨论**

注意，类包括枚举（8.9节），接口包括注释类型（9.6节）。

注释还可以用在枚举常量上。这样的注释直接放在它们注释的枚举常量前面。如果通过用于给定注释类型的多个注释来为声明加注释，则会发生编译时错误。

**讨论**

注释传统上置于所有其他修饰符之前，但这不是一种要求；它们可以自由地混合在其他修饰符中间。

注释有三类。第一类（正常注释）非常普通。另外两类（标记注释和单元素注释）只是简略表示。

*Annotations:*

*Annotation*

*Annotations Annotation*

*Annotation:*

*NormalAnnotation*

*MarkerAnnotation*

*SingleElementAnnotation*

正常注释用于注释程序元素：

*NormalAnnotation:*

*@ TypeName ( ElementValuePairs<sub>opt</sub> )*

*ElementValuePairs:*

*ElementValuePair*

*ElementValuePairs , ElementValuePair*

*ElementValuePair:*

*Identifier = ElementValue*

*ElementValue:*

*ConditionalExpression*

*Annotation*

*ElementValueArrayInitializer*

*ElementValueArrayInitializer:*

*{ ElementValues<sub>opt</sub> ,opt }*

*ElementValues:*

*ElementValue*

*ElementValues , ElementValue*

**讨论**

注意：at 符号 (@) 是一个到达自身的标记。从技术上讲，可以在 at 符号与 *TypeName* 之间加空格，但是不鼓励这样做。

*TypeName* 指定了与注释对应的注释类型。如果 *TypeName* 没有指定一个注释类型，则会发生编译时错误。由注释指定的注释类型在使用注释那一点必须是可访问的 (6.6 节)，否则就会发生编译时错误。

*ElementValuePair* 中的 *Identifier* 必须是注释类型的元素之一的简单名称，该注释类型是由包含的注释中的 *TypeName* 标识的。否则，就会发生编译时错误 (换句话说，元素-值对中的标识符还必须是通过 *TypeName* 标识的接口中的方法名称)。

这个方法的返回类型定义了元素-值对的元素类型。*ElementValueArrayInitializer* 类似于正常的数组初始化语句 (10.6 节)，只不过允许注释代替表达式。

当且仅当下列条件之一成立时，元素类型 *T* 才与元素值 *V* 匹配：

- *T* 是数组类型 *E*[], 并且下列两个条件之一成立：
  - ◆ *V* 是一个 *ElementValueArrayInitializer*，并且 *V* 中的每个 *ElementValueInitializer* (类似于数组初始化语句中的变量初始化语句) 与 *E* 匹配。
  - ◆ *V* 是一个 *ElementValue*，它与 *T* 匹配。
- *V* 的类型与 *T* 是赋值兼容的 (5.2 节)，此外：
  - ◆ 如果 *T* 是基本类型或 *String*，*V* 是常量表达式 (15.28 节)。
  - ◆ *V* 不为空。
  - ◆ 如果 *T* 是 *Class* 或者 *Class* 的调用，并且 *V* 是类常量 (15.8.2 节)。
  - ◆ 如果 *T* 是枚举类型，并且 *V* 是枚举常量。

如果元素类型与 *ElementValue* 不匹配，则会发生编译时错误。

如果元素类型不是注释类型或数组类型，则 *ElementValue* 必须是一个 *Conditional-Expression* (15.25 节)。

**讨论**

注意，*null* 不是任何元素类型合法的元素值。

如果元素类型是数组类型，并且对应的 *ElementValue* 不是一个 *ElementValueArrayInitializer*，就会将一个其惟一元素是由 *ElementValue* 表示的值的数组值与元素相关联。否则，将由 *ElementValue* 表示的值与元素相关联。

**讨论**

换句话说，当把单元素数组与数组赋值的注释类型元素相关联时，允许省略花括号。注意，数组的元素类型不能是数组类型，也就是说，不允许把嵌套数组类型用作元素

类型（虽然注释语法允许这样做，但是注释类型声明语法则不允许）。

注释类型声明上的注释被称为元注释。注释类型可用于注释它自己的声明。更一般地讲，允许“注释”关系的传递闭包中的循环性。例如，用一个注释类型来注释另一个注释类型声明以及用后一个类型来注释前一个类型声明是合法的（预定义的元注释类型包含多种这样的循环性）。

#### 例程

下面是正常注释的一个示例：

```
// Normal annotation
@RequestForEnhancement(
    id          = 2868724,
    synopsis    = "Provide time-travel functionality",
    engineer     = "Mr. Peabody",
    date        = "4/1/2004"
)
public static void travelThroughTime(Date destination) { ... }
```

注意，本节中的示例中的注释类型都是第 9.6 节中定义的注释类型。另请注意，上述注释中的元素与对应的注释类型声明中的元素顺序相同。不要求这样做，但是除非特定的环境规定了另外的情况，否则这是要遵守的合理约定。

注释的第二种形式即标记注释是一种简略表示，设计用于和标记注释类型一起使用：

*MarkerAnnotation:*

@ *TypeName*

它只是正常注释的一种简略表示：

@*TypeName*()

示例：

```
// Marker annotation
@Preliminary public class TimeTravel { ... }
```

注意，只要所有的元素都具有默认值，那么为具有这些元素的注释类型使用标记注释就是合法的。

注释的第三种形式即单元素注释是一种简略表示，设计用于和单元素注释类型一起使用：

*SingleElementAnnotation:*

@ *TypeName* ( *ElementValue* )

它是正常注释的一种简略表示：

```
@TypeName ( value = ElementValue )
```

## 讨论

示例：

```
// Single-element annotation
@Copyright("2002 Yoyodyne Propulsion Systems, Inc., All rights reserved.")
public class OscillationOverthruster { ... }
```

具有数组赋值的单元素注释的示例：

```
// Array-valued single-element annotation
@Endorsers({"Children", "Unscrupulous dentists"})
public class Lollipop { ... }
```

具有单元素数组赋值的单元素注释的示例（注意，省略了花括号）：

```
// Single-element array-valued single-element annotation
@Endorsers("Epicurus")
public class Pleasure { ... }
```

具有复杂注释的示例：

```
// Single-element complex annotation
@author(@Name(first = "Joe", last = "Hacker"))
public class BitTwiddle { ... }
```

注意，只要其中一个元素是指定的值，并且所有其他元素具有默认值，那么为具有多个元素的注释类型使用单元素注释就是合法的。

下面是一个利用默认值的注释的示例：

```
// Normal annotation with default values
@RequestForEnhancement(
    id = 4561414,
    synopsis = "Balance the federal budget"
)
public static void balanceFederalBudget() {
    throw new UnsupportedOperationException("Not implemented");
}
```

下面是一个具有 Class 元素的注释的示例，通过使用有界通配符来限制该元素的值。

```
// Single-element annotation with Class element restricted by bounded wildcard
// The annotation presumes the existence of this class.
class GorgeousFormatter implements Formatter { ... }
@PrettyPrinter(GorgeousFormatter.class) public class Petunia {...}
// This annotation is illegal, as String is not a subtype of Formatter!!
```

```
@PrettyPrinter(String.class) public class Begonia { ... }
```

下面是一个使用枚举类型的注释的示例，该枚举类型定义在注释类型的内部：

```
// Annotation using enum type declared inside the annotation type
@Quality(Quality.Level.GOOD)
public class Karma {
    ...
}
```

---

死亡、生命和睡眠、现实和思想，  
帮帮我，上帝，我想知道它们的界限……  
——William Wordsworth, 《Maternal Grief》



就是所罗门极荣华的时候，他所穿戴的还不如这一朵花呢。

——《马太福音》第 6 章第 29 节

在 Java 编程语言中，数组是对象（4.3.1 节），是动态创建的，并且可以赋值给 Object 类型（4.3.2 节）的变量。类 Object 的所有方法都可以在数组上被调用。

一个数组对象包含许多变量。变量的个数可能为零，在这种情况下，就称数组为空数组。数组中包含的变量没有名称；作为替代，通过使用非负整数下标值的数组访问表达式来引用它们。这些变量被称为数组的元素。如果一个数组具有  $n$  个元素，就称  $n$  是数组的长度；使用  $0 \sim n-1$ （包含  $n-1$ ）的整数下标来引用数组的元素。

数组的所有元素都有相同的类型，称为数组的元素类型。如果数组的元素类型是  $T$ ，则将数组本身的类型写作  $T[]$ 。

`float` 类型的数组元素的值总是浮点值集（4.2.3 节）的元素；类似地，`double` 类型的数组元素的值总是双精度值集的元素。`float` 类型的数组元素的值不允许作为非浮点值集元素的浮点扩展的指数值集的元素，`double` 类型的数组元素的值也不允许作为非双精度值集元素的双精度扩展的指数值集的元素。

数组的元素类型本身可能是一种数组类型。这样一个数组的元素可以包含指向子数组的引用。从任意数组类型出发，如果人们考虑其元素类型，然后（如果那也是一种数组类型）考虑那种类型的元素类型，依此类推，最终人们必须到达一种不是数组类型的元素类型（component type），这称为原始数组的元素类型（element type），在数据结构这个级别上，把这些元素（component）称为原始数组的元素（element）。

有些情况下，数组的元素可以是一个数组：如果元素类型是 `Object`、`Cloneable` 或 `java.io.Serializable`，则其中一些或所有元素都可能是数组，因为任何数组对象都可以被赋值给这些类型的任何变量。

## 10.1 数组类型

数组类型可以写成：元素类型的名称后接若干对空方括号[]。括号对的数量指定了数

组嵌套的深度。数组的长度不是其类型的一部分。

数组的元素类型可以是任何类型，包括基本类型或引用类型，特别地：

- 允许数组把接口类型作为其元素类型。这样一个数组的元素可以把空引用或者实现该接口的任何类型的实例作为它们的值。
- 允许数组把 `abstract` 类类型作为其元素类型。这样一个数组的元素可以把空引用或者自身不是 `abstract` 的 `abstract` 类的任何子类的实例作为它们的值。

数组类型可以用在声明和强制类型转换表达式（15.16 节）中。

## 10.2 数组变量

数组类型的变量存储一个指向对象的引用。声明数组类型的变量不会创建一个数组对象，或者为数组元素分配任何空间。它只会创建变量本身，该变量可以包含一个指向数组的引用。但是，声明符（8.3 节）的初始化语句部分可以创建一个数组，这样，指向该数组的引用将成为变量的初始值。

由于数组的长度不是其类型的一部分，因此数组类型的单个变量可以包含指向具有不同长度的数组的引用。

下面是一个数组变量声明的示例，它不会创建数组：

```
int[] ai;           // array of int
short[][] as;       // array of array of short
Object[] ao,        // array of Object
        otherAo;    // array of Object
Collection<?>[] ca; // array of Collection of unknown type
short s,            // scalar short
        aas[][];    // array of array of short
```

下面是数组变量声明的一些示例，它们会创建数组对象：

```
Exception ae[] = new Exception[3];
Object aao[][] = new Exception[2][3];
int[] factorial = { 1, 1, 2, 6, 24, 120, 720, 5040 };
char ac[] = { 'n', 'o', 't', ' ', 'a', ' ', 'S', 't', 'r', 'i', 'n', 'g' };
String[] aas = { "array", "of", "String", " "};
```

[] 可以作为类型的一部分出现在声明的开始处，或者作为特殊变量的声明符的一部分，或者同时作为这二者，如在下面的示例中：

```
byte[] rowvector, colvector, matrix[];
```

该声明等价于：

```
byte rowvector[], colvector[], matrix[][];
```

一旦创建了数组对象，则其长度永远也不会发生改变。为了使数组变量引用不同长度的数组，必须把指向不同数组的引用赋予该变量。

如果数组变量 `v` 具有类型 `A[]`，其中 `A` 是引用类型，那么 `v` 可以存储一个指向任意数组类型 `B[]` 的实例的引用，假定可以把 `B` 赋予 `A`。这可能在以后的赋值中导致运行时异常：

相关讨论请参见第 10.10 节。

## 10.3 数组创建

通过数组创建表达式（15.10 节）或数组初始化语句（10.6 节）来创建数组。

数组创建表达式指定了元素类型、嵌套数组的层数以及至少一个嵌套层的数组长度。数组的长度可作为最终实例变量 `length` 使用。如果元素类型不是可具体化的类型（4.7 节），则会发生编译时错误。

数组初始化语句会创建一个数组，并为其所有元素提供初始值。

## 10.4 数组访问

数组的元素可以被数组访问表达式（15.13 节）访问，该表达式包含一个其值为数组引用的表达式，其后接有通过 `[` 和 `]` 封闭的下标表达式，如 `A[i]`。所有的数组都以 0 为起点。长度为  $n$  的数组可以把整数  $0 \sim n-1$  作为其下标。

数组的下标必须是 `int` 值；`short`、`byte` 或 `char` 值也可以用作下标值，这是由于可以对它们进行一元数值提升，使之成为 `int` 值。如果试图利用 `long` 下标值访问数组元素，则会导致编译时错误。

在运行时会检查所有的数组访问：如果试图使用小于 0 或者大于或等于数组长度的下标，则会导致抛出一个 `ArrayIndexOutOfBoundsException`。

## 10.5 数组：一个简单的示例

示例：

```
class Gauss {
    public static void main(String[] args) {
        int[] ia = new int[101];
        for (int i = 0; i < ia.length; i++)
            ia[i] = i;
        int sum = 0;
        for (int e : ia)
            sum += e;
        System.out.println(sum);
    }
}
```

产生如下输出：

5050

声明一个变量 `ia`，其类型为 `int` 数组，即 `int[]`。将变量 `ia` 初始化为一个最新创建的数组对象，它是通过数组创建表达式（15.10 节）创建的。数组创建表达式指定数组

应该具有 101 个元素。正如所显示的那样，可以通过字段 `length` 来使用数组的长度。

示例程序用 0~100 之间的整数来填充数组，对这些整数求和，并输出结果。

## 10.6 数组初始化语句

可以在声明中指定数组初始化语句，或者将其作为数组创建表达式（15.10 节）的一部分，用于创建数组并提供一些初始值：

*ArrayInitializer:*

*{ VariableInitializers<sub>opt</sub> ,opt }*

*VariableInitializers:*

*VariableInitializer*

*VariableInitializers , VariableInitializer*

下面重复了第 8.3 节的内容，以使这里的介绍更清楚：

*VariableInitializer:*

*Expression*

*ArrayInitializer*

数组初始化语句可以写成用逗号分隔的表达式列表，用括号“{”和“}”将其封闭起来。构造的数组长度等于表达式的数量。数组初始化语句中的表达式按照它们出现在源代码中的顺序从左到右依次执行。第  $n$  个变量初始化语句指定第  $n-1$  个数组元素。每个表达式与数组的元素类型必须是赋值兼容的（5.2 节），否则就会发生编译时错误。如果被初始化的数组的元素类型不是可具体化的（4.7 节），则会发生编译时错误。

如果元素类型本身是数组类型，那么指定元素的表达式本身将是数组初始化语句；也就是说，数组初始化语句可能是嵌套的。

在数组初始化语句中的最后一个表达式后面可能出现一个尾随的逗号，可将其忽略。例如：

```
class Test {
    public static void main(String[] args) {
        int ia[][] = { {1, 2}, null };
        for (int[] ea : ia)
            for (int e: ea)
                System.out.println(e);
    }
}
```

输出：

1  
2

在尝试访问数组 `ia` 的第二个元素的下标时会引发一个 `NullPointerException`，因为它是一个空引用。

## 10.7 数组成员

数组类型的成员包括以下所有成员：

- `public final` 字段 `length`，它包含数组的元素个数（`length` 可能为正数或 0）。
- `public` 方法 `clone`，它重写了类 `Object` 中的同名方法，并且不会抛出受查异常。

数组类型 `T[]` 的 `clone` 方法的返回类型是 `T[]`。

- 从类 `Object` 继承的所有成员；惟一未被继承的 `Object` 的方法是其 `clone` 方法。
- 因此，数组具有与以下类相同的公共字段和方法：

```
class A<T> implements Cloneable, java.io.Serializable {
    public final int length = X;
    public T[] clone() {
        try {
            return (T[])super.clone(); // unchecked warning
        } catch (CloneNotSupportedException e) {
            throw new InternalError(e.getMessage());
        }
    }
}
```

注意，如果确实以这种方式实现数组，则上述示例中的强制类型转换将生成一个未经检查的警告（5.1.9 节）。

每个数组都会实现接口 `Cloneable` 和 `java.io.Serializable`。

如测试程序所显示的那样，数组是可复制的：

```
class Test {
    public static void main(String[] args) {
        int ia1[] = { 1, 2 };
        int ia2[] = ia1.clone();
        System.out.print((ia1 == ia2) + " ");
        ia1[1]++;
        System.out.println(ia2[1]);
    }
}
```

输出如下：

```
false 2
```

这表明被 `ia1` 和 `ia2` 引用的数组元素是不同的变量（在 Java 编程语言的某些早期实现中，这个实例无法通过编译，这是由于编译器会错误地认为数组的 `clone` 方法可能抛出一个 `CloneNotSupportedException`）。

多维数组的 `clone` 方法是一种浅复制，也就是说，它只会创建单独一个新数组。子数组是共享的。下面的示例程序显示了这一点：

```
class Test {
    public static void main(String[] args) throws Throwable {
        int ia[][] = { { 1, 2 }, null };
        int ja[][] = ia.clone();
    }
}
```

```
        System.out.print((ia == ja) + " ");
        System.out.println(ia[0] == ja[0] && ia[1] == ja[1]);
    }
}
```

输出如下：

```
false true
```

这表明 `int[]` 数组 `ia[0]` 和 `int[]` 数组 `ja[0]` 是相同的数组。

## 10.8 数组的 Class 对象

每个数组都有一个关联的 `Class` 对象，它被具有相同元素类型的所有其他数组共享。数组类型的直接超类是 `Object`。每个数组类型都实现了接口 `Cloneable` 和 `java.io.Serializable`。

下面的示例代码显示了这一点：

```
class Test {
    public static void main(String[] args) {
        int[] ia = new int[3];
        System.out.println(ia.getClass());
        System.out.println(ia.getClass().getSuperclass());
    }
}
```

输出如下：

```
class [I
class java.lang.Object
```

其中字符串 “[I” 是类对象 “元素类型为 `int` 的数组” 的运行时类型签名。

## 10.9 字符的数组不是一个 String

与 C 不同的是，在 Java 编程语言中，`char` 的数组不是一个 `String`，`String` 和 `char` 的数组都不是以 ‘\u0000’（NUL 字符）终止的。

`String` 对象是恒定不变的，也就是说，其内容永远不会改变，尽管 `char` 的数组具有可变的元素。类 `String` 中的方法 `toCharArray` 返回一个字符的数组，其中包含有与 `String` 相同的字符序列。类 `StringBuffer` 实现了字符的可变数组上的有用方法。

## 10.10 数组存储异常

如果一个数组变量 `v` 具有类型 `A[]`，其中 `A` 是一种引用类型，则 `v` 可以存储一个指向任意数组类型 `B[]` 的实例的引用，假定可以把 `B` 赋值给 `A`。

因此，下面的示例：



```
class Point { int x, y; }
class ColoredPoint extends Point { int color; }
class Test {
    public static void main(String[] args) {
        ColoredPoint[] cpa = new ColoredPoint[10];
        Point[] pa = cpa;
        System.out.println(pa[1] == null);
        try {
            pa[0] = new Point();
        } catch (ArrayStoreException e) {
            System.out.println(e);
        }
    }
}
```

产生如下输出：

```
true
java.lang.ArrayStoreException
```

这里的变量 `pa` 具有类型 `Point[]`，变量 `cpa` 将一个指向类型 `ColoredPoint[]` 的对象的引用作为它的值。可以把一个 `ColoredPoint` 赋值给一个 `Point`；因此，可以把 `cpa` 的值赋予 `pa`。

例如，在测试 `pa[1]` 是否为 `null` 时，指向这个数组 `pa` 的引用不会导致一个运行时类型错误。这是由于类型 `ColoredPoint[]` 的数组的元素是一个 `ColoredPoint`，并且由于 `Point` 是 `ColoredPoint` 的超类，所以每个 `ColoredPoint` 都可以代替一个 `Point`。

另一方面，给数组 `pa` 赋值可能导致一个运行时错误。在编译时，将会检查对 `pa` 的元素赋值，以确保所赋予的值是一个 `Point`。但是，由于 `pa` 存储了一个指向 `ColoredPoint` 的数组的引用，因此更确切地讲，仅当在运行时所赋予的值的类型是一个 `ColoredPoint` 时，赋值才有效。

Java 虚拟机在运行时会检查这种情形，以确保赋值是有效的；如果不是，则会抛出一个 `ArrayStoreException`。更正式地讲：在运行时将会检查对类型为 `A[]` 的数组元素的赋值（其中 `A` 是一种引用类型），以确保所赋予的值可以赋予数组的实际元素类型，其中实际元素类型可以是可赋予 `A` 的任何引用类型。

## 讨论

如果数组的元素类型不是可具体化的（4.7 节），则虚拟机将不能执行上一个段落中描述的存储检查。这就是为什么禁止创建不可具体化类型的数组的原因。程序员可以声明其元素类型不可具体化的数组类型的变量，但是如果试图给它们赋值，则会引发一个未经检查的警告（5.1.9 节）。

最后在银色的狂欢中爆破，  
带着羽毛、冠状头饰和所有华贵的衣服……  
——John Keats, 《The Eve of St. Agnes》(1819)

# 第 11 章

## 异 常

会出错的，终将会出错。

——菲纳格定律

（通常误认为这是墨菲说的，实际上二者有很大差别——这里只是为了显示菲纳格是正确的）

当程序违反了 Java 编程语言的语义约束时，Java 虚拟机就会把这个错误作为一个异常发送给程序。一个这样的异常示例是试图使用数组边界之外的下标。有些编程语言及其实现通过紧急终止程序对这类错误做出反应；另外一些编程语言则允许实现以一种随意或不可预测的方式做出反应。这些方法与 Java 平台提供可移植性和健壮性的设计目标都不兼容。相反，Java 编程语言规定：当违反语义约束时将会抛出一个异常，并将引起控制从发生异常的那一点非局部转移到可以由程序员指定的那一点。我们称异常是从它发生的那一点抛出的，并在控制转移到的那一点被捕获。

程序也可以使用 `throw` 语句（14.18 节）显式地抛出异常。

显式使用 `throw` 语句对通过返回古怪的值（例如，在通常不希望返回负值的地方，返回整型值 -1）处理错误条件的旧式编程风格提供了一种替代选择。经验表明：这样的古怪值通常会被调用方忽略或者不会对其进行检查，从而导致程序不是健壮的，展现了不合需要的行为，或者二者兼而有之。

每个异常都是通过类 `Throwable` 或其子类之一的实例表示的；这样一个对象可用于携带从发生异常的那一点到捕获它的处理程序之类的信息，处理程序是由 `try` 语句（14.20 节）的 `catch` 子句建立的。在抛出异常过程中，Java 虚拟机会突然一个接一个地完成在当前线程中已开始执行但尚未完成的任何表达式、语句、方法和构造函数调用、初始化语句以及字段初始化表达式。这个过程会继续，直到发现一个处理程序指示它通过命名异常的类或者异常的类的超类，而处理了那个特定的异常。如果没有发现这样一个处理程序，那么就会为当前线程的父线程 `ThreadGroup` 调用方法 `uncaughtException`——因此，所做的任何努力都是为了避免使一个异常未被处理。

Java 平台的异常机制与其同步模型（第 17 章）集成在一起，以使锁被释放为 `synchronized` 语句（14.19 节），并且 `synchronized` 方法（8.4.3.6 节、15.12 节）的调用会突然完成。

本章描述了异常的不同起因（11.1 节），详细说明了异常在编译时是如何检查的（11.2 节）以及在运行时是如何处理的（11.3 节）。然后给出了一个详细的示例（11.4 节），接着解释了异常的层次结构（11.5 节）。

## 11.1 异常的起因

如果我们不成功，那么就会承担失败的风险。  
——J. Danforth Quayle（美国第 44 任副总统）

出于以下三种原因之一会抛出一个异常：

- Java 虚拟机同步检测到一个异常执行条件。这种条件是由于以下原因引起的：
  - ◆ 表达式的求值违反了语言的正常语义，例如，整数除以 0，参见第 15.6 节中总结的情况。
  - ◆ 在加载或链接程序的某一部分时出错（12.2 节、12.3 节）。
  - ◆ 超过了资源上的某些限制，例如，使用了太多的内存。这些异常不会在程序中的任意点抛出，而是在将它们指定为表达式求值或语句执行的可能结果的那一点抛出。
- 执行 `throw` 语句（14.18 节）。
- 由于以下原因之一而发生异步异常：
  - ◆ 调用了类 `Thread`（已否决的）方法 `stop`。
  - ◆ 虚拟机中发生了一个内部错误（11.5.2 节）。

异常是由类 `Throwable` 的实例及其子类的实例表示的。这些类统称为异常类。

## 11.2 异常的编译时检查

Java 编程语言的编译器通过分析哪些受查异常可能是由方法或构造函数的执行产生的，在编译时检查程序包含有针对受查异常的处理程序。对于作为可能结果的每个受查异常，方法（8.4.6 节）或构造函数（8.8.5 节）的 `throws` 子句必须提及那个异常的类，或者那个异常的类的超类之一。这种针对异常处理程序的存在情况的编译时检查设计用于减少未被正确处理的异常数量。

未经检查的异常类（*unchecked exceptions class*）包括类 `RuntimeException` 和其子类，以及类 `Error` 和其子类。所有其他的异常类都是受查异常类（*checked exception class*）。Java API 定义了许多异常类，既有受查异常类，也有未经检查的异常类。程序员可以另外声明受查和未经检查的异常类。有关异常类层次结构以及 Java API 和 Java 虚拟机定义的某些异常类的说明，请参见 11.5 节。

`throws` 子句中指定的受查异常类是方法或构造函数的实现者与用户之间契约的一部分。对于任何受查异常，如果被重写的方法不允许通过其 `throws` 子句抛出它们，则重写方法的 `throws` 子句不能指定该方法将导致抛出这些异常。当涉及接口时，可能通过单独

一条重写声明来重写多个方法声明。在这种情况下，重写的声明必须具有一条 `throws` 子句，它与所有被重写的声明兼容（9.4 节）。

依据下面给出的规则，如果语句或表达式的执行可能导致类型  $E$  的异常被抛出，则称该语句或表达式可以抛出一个受查异常类型  $E$ 。

### 11.2.1 表达式的异常分析

当且仅当下列条件之一成立时，方法调用表达式可以抛出一个异常类型  $E$ ：

- 将被调用的、形如 *Primary.Identifier* 和 *Primary* 表达式的方法可以抛出  $E$ 。
- 参数列表的某个表达式可以抛出  $E$ 。
- $E$  在被调用方法的类型的 `throws` 子句中列出。

当且仅当下列条件之一成立时，类实例创建表达式可以抛出一个异常类型  $E$ ：

- 表达式是限定的类实例创建表达式，并且限定的表达式可以抛出  $E$ 。
- 参数列表的某个表达式可以抛出  $E$ 。
- $E$  在被调用构造函数的类型的 `throws` 子句中列出。
- 类实例创建表达式包括一个 *ClassBody*，并且该 *ClassBody* 中的某个实例初始化语句块或实例变量初始化表达式可以抛出  $E$ 。

对于所有其他类型的表达式，当且仅当它的直接子表达式之一可以抛出  $E$  时，该表达式才能抛出类型  $E$ 。

### 11.2.2 语句的异常分析

当且仅当 `throw` 表达式的静态类型为  $E$  或  $E$  的子类型时，`throw` 语句才会抛出一个异常类型  $E$ ，否则 `throw` 表达式可以抛出  $E$ 。

当且仅当以下条件之一成立时，显式构造函数调用语句才可以抛出一个异常类型  $E$ ：

- 构造函数调用的参数列表的某个子表达式可以抛出  $E$ 。
- 在被调用的构造函数的 `throws` 子句中声明了  $E$ 。

当且仅当以下条件之一成立时，`try` 语句才可以抛出一个异常类型  $E$ ：

- `try` 语句块可以抛出  $E$ ，并且  $E$  不能赋予 `try` 语句的任何 `catch` 参数，此外，`finally` 语句块不存在，或者 `finally` 语句块可以正常完成。
- `try` 语句的某个 `catch` 语句块可以抛出  $E$ ，并且 `finally` 语句块不存在，或者 `finally` 语句块可以正常完成。
- `finally` 语句块存在并能抛出  $E$ 。

当且仅当任何其他语句  $S$  中直接包含的表达式或语句可以抛出一个异常类型  $E$  时， $S$  才可以抛出  $E$ 。

### 11.2.3 异常检查

当以下两个条件都成立时，如果方法体或构造函数体可以抛出某个异常类型  $E$ ，则会

发生编译时错误：

- $E$  是受查异常类型。
- $E$  不是方法或构造函数的 `throws` 子句中声明的某个类型的子类型。

如果命名类或接口（8.3.2 节）内的静态初始化语句（8.7 节）或类变量初始化语句可以抛出一个受查异常类型，则会发生编译时错误。

如果命名类的实例变量初始化语句可以抛出一个受查异常，那么除非该异常或其子类型之一是其类的每个构造函数的 `throws` 子句中显式声明的，并且该类至少具有一个显式声明的构造函数，否则就会发生编译时错误。匿名类（15.9.5 节）中的实例变量初始化语句可以抛出任何异常。

如果 `catch` 子句可以捕获受查异常类型  $E1$ ，但是不存在使下列所有条件均成立的受查异常类型  $E2$ ，那么除非  $E1$  是类 `Exception`，否则会发生编译时错误：

- $E2 <: E1$
- 对应于 `catch` 子句的 `try` 语句块可以抛出  $E2$ 。
- 没有直接封闭的 `try` 语句的前置 `catch` 语句块可用于捕获  $E2$  或  $E2$  的子类型。

#### 11.2.4 为什么错误未被检查

对于那些是错误类（`Error` 及其子类）的未经检查的异常类，将免除对其进行编译时检查，这是因为它们会出现在程序中的许多点处，并且难以或不可能从其恢复。声明这种异常的程序将是混乱的，没有实际意义。

#### 11.2.5 为什么运行时异常未被检查

运行时异常类（`RuntimeException` 及其子类）将被免除进行编译时检查，这是由于 Java 编程语言的设计者判断，声明这类异常不会对确保程序的正确性带来显著的帮助。Java 编程语言的许多操作和构造可能导致运行时异常。编译器可用的信息以及编译器执行的分析级别通常不足以确保这类运行时异常不会发生，即使这对于程序员是明显的也是如此。要求声明这类异常类只会激怒程序员。

例如，某些代码可能实现一个循环数据结构，根据构造，它可能永远不会涉及 `null` 引用；这样，程序员就可以确定 `NullPointerException` 不会发生，但是编译器却难以证明这一点。建立数据结构的这种全局属性需要经过定理证明的技术，它超出了本规范的范围。

### 11.3 异常处理

当抛出异常时，就将控制从引发异常的代码转移到处理异常的 `try` 语句（14.20 节）中最近的动态封闭的 `catch` 子句。

如果语句或表达式出现在 `try` 语句（`catch` 语句是其一部分）的 `try` 语句块内，或者如果语句或表达式的调用者被 `catch` 子句动态封闭，则语句或表达式就是被 `catch` 子



句动态封闭的。

语句或表达式的调用方依赖于其出现的位置：

- 如果在方法内，那么调用方就是方法调用表达式（15.12 节），执行它可以调用方法。
- 如果在构造函数、实例初始化语句或实例变量的初始化语句内，那么调用方就是类实例创建表达式（15.9 节）或者 `newInstance` 的方法调用，执行它可以创建对象。
- 如果在静态初始化语句或者 `static` 变量的静态初始化语句内，那么调用方就是表达式，它使用类或接口以使其被初始化。

至于特定的 `catch` 子句是否会处理异常，这是通过比较被抛出的对象的类与 `catch` 子句的参数的声明类型来确定的。如果 `catch` 子句的参数的类型是异常的类或者异常类的超类，那么 `catch` 子句就会处理异常。换句话说，`catch` 子句将捕获其声明的参数类型是 `instanceof`（15.20.2 节）的任何异常对象。

发生在抛出异常时的控制转移会引起表达式（15.6 节）和语句（14.1 节）的突然完成，直至遇到一个可以处理异常的 `catch` 子句；然后通过执行 `catch` 子句的语句块而使执行继续进行下去。引发异常的代码永远不会被重新执行。

如果没有找到处理异常的 `catch` 子句，那么当前线程（遇到异常的线程）就会终止，但是这仅出现在执行了所有的 `finally` 语句，并且已经为当前线程的父线程 `ThreadGroup` 调用了方法 `uncaughtException` 之后。

在某些情况下，人们希望确保一个代码块总是在另一个代码块之后执行，即使另一个代码块突然完成也是如此，在这种情况下，可以使用带 `finally` 子句（14.20.2 节）的 `try` 语句。

如果 `try-finally` 或 `try-catch-finally` 语句中的 `try` 或 `catch` 语句块突然完成，那么就会在传播异常期间执行 `finally` 子句，即使最终没有找到匹配的 `catch` 子句也是如此。如果由于 `try` 语句块突然完成而执行 `finally` 子句，并且 `finally` 子句自身也突然完成，那么就会丢弃 `try` 语句块突然完成的原因，并从那里传播新的突然完成的原因。

第 14 章中每一条语句的规范以及第 15 章（特别是第 15.6 节）中表达式的规范详细说明了突然完成和异常捕获的准确规则。

### 11.3.1 异常是精确的

异常是精确的：当发生控制转移时，在抛出异常那一点之前，语句执行和表达式求值的效果看上去必须已经发生。在抛出异常那一点之后，从此出现的表达式、语句或代码部分似乎都不会被求值。如果优化的代码冒险地执行一些接在发生异常那一点之后的表达式或语句，那么这种代码必须准备好把这种冒险的执行对用户可见的程序状态隐藏起来。

### 11.3.2 处理异步异常

大多数异常是同步发生的，这是由在其中发生异常的线程的动作引起的；另外它们还是在程序中某一点处发生的，这一点是这种异常中指定的可能结果。与之对照，异步异常是可能在程序执行中的任何点发生的异常。



如果要生成高质量的机器码，那么就有必要正确理解异步异常的语义。

异步异常很少见，仅当以下条件之一成立时，它们才会发生：

- 调用类 Thread 或 ThreadGroup 的 stop 方法
- Java 虚拟机中出现内部错误（11.5.2 节）

一个线程可能会调用 stop 方法，以影响指定线程组中的另一个线程或所有线程。它们是异步的，这是由于它们可能出现在另一个线程或其他线程执行中的任意点。

InternalError 被认为是异步的。

Java 平台允许在抛出异步异常之前，执行一小部分数量有限的语句。Java 平台允许这种延迟，以便优化的代码检测到异步异常，并在用于处理它们的那一点抛出这些异常，同时遵守 Java 编程语言的语义。

一个简单的实现可以在每个控制转移指令那一点处轮询异步异常。由于程序大小有限，这就给检测异步异常的总延迟提供了一个界限。由于在控制转移之间不会发生异步异常，代码生成器就具有在控制转移之间重新排列计算顺序的某种灵活性，以便获得更高的性能。

建议进一步阅读 Marc Feeley 所著的论文《Polling Efficiently on Stock Hardware》，收录于《Proc. 1993 Conference on Functional Programming and Computer Architecture》，丹麦哥本哈根，第 179~187 页。

与所有异常一样，异步异常也是精确的（11.3.1 节）。

## 11.4 异常的示例

考虑以下示例：

```
class TestException extends Exception {
    TestException() { super(); }
    TestException(String s) { super(s); }
}
class Test {
    public static void main(String[] args) {
        for (String arg : args) {
            try {
                thrower(arg);
                System.out.println("Test \"" + arg +
                    "\" didn't throw an exception");
            } catch (Exception e) {
                System.out.println("Test \"" + arg +
                    "\" threw a " + e.getClass() +
                    "\n with message: " + e.getMessage());
            }
        }
    }
    static int thrower(String s) throws TestException {
        try {
            if (s.equals("divide")) {
```

```
        int i = 0;
        return i/i;
    }
    if (s.equals("null")) {
        s = null;
        return s.length();
    }
    if (s.equals("test"))
        throw new TestException("Test message");
    return 0;
} finally {
    System.out.println("[thrower(\"" + s +
        "\") done]");
}
}
```

如果执行该程序，将以下参数传递给它：

```
divide null not test
```

则会产生如下输出：

```
[thrower('divide') done]
Test 'divide' threw a class java.lang.ArithmeticException
    with message: / by zero
[thrower('null') done]
Test 'null' threw a class java.lang.NullPointerException
    with message: null
[thrower('not') done]
Test 'not' didn't throw an exception
[thrower('test') done]
Test 'test' threw a class TestException
    with message: Test message
```

本示例声明了一个异常类 `TestException`。类 `Test` 的 `main` 方法调用 `thrower` 方法 4 次，引起异常被抛出 3 或 4 次。`main` 方法中的 `try` 语句捕获 `thrower` 抛出的每个异常。无论 `thrower` 的调用是正常完成，还是突然完成，都会打印一条消息描述发生了什么情况。

方法 `thrower` 的声明必须具有一条 `throws` 子句，这是因为它能够抛出 `TestException` 的实例，`TestException` 是一个受查异常类（11.2 节）。如果遗漏了 `throws` 子句，则会发生编译时错误。

注意，在每次调用 `thrower` 时，都会执行 `finally` 子句，而不管异常是否发生，如每次调用的输出 “[thrower (...) done]” 所示。

## 11.5 异常层次结构

程序中可能的异常被组织进类的层次结构中，其根为类 `Throwable`（11.5 节），它是 `Object` 的直接子类。类 `Exception` 和 `Error` 是 `Throwable` 的直接子类。类

`RuntimeException` 是 `Exception` 的直接子类。

程序可以使用 `throw` 语句中预先存在的异常类，或者根据需要，把额外的异常类定义成 `Throwable` 或其任何子类的子类。为了利用 Java 平台的异常处理程序的编译时检查功能，通常把大多数新的异常类定义成受查异常类，特别是把它们定义成 `Exception` 的子类，而不是 `RuntimeException` 的子类。

类 `Exception` 是普通程序可能希望从其恢复的所有异常的超类。类 `RuntimeException` 是类 `Exception` 的子类。`RuntimeException` 的子类是未受查异常类。`The subclasses of Exception`（而不是 `RuntimeException`）的子类及其子类都是受查异常类。

类 `Error` 及其子类都是普通程序不能像平常那样指望从其中恢复的异常。参见 Java API 规范，以了解异常层次结构的详细描述。

类 `Error` 是 `Throwable` 单独的子类（不同于类层次结构中的 `Exception`），它允许程序使用以下语法：

```
    } catch (Exception e) {
```

来捕获可能从其恢复的所有异常，而不会捕获通常不可能从其恢复的错误。

### 11.5.1 加载和链接错误

当发生加载、链接、准备、验证或初始化错误时，Java 虚拟机就会抛出一个对象，它是 `LinkageError` 的子类的实例：

- 第 12.2 节中描述了加载过程。
- 第 12.3 节中描述了链接过程。
- 第 12.3.1 节中描述了类验证过程。
- 第 12.3.2 节中描述了类准备过程。
- 第 12.4 节中描述了类初始化过程。

### 11.5.2 虚拟机错误

当内部错误或资源限制阻止 Java 虚拟机实现 Java 编程语言的语义时，Java 虚拟机就会抛出一个对象，它是类 `VirtualMachineError` 的子类的实例。有关这些错误的权威讨论，请参见《The Java™ Virtual Machine Specification Second Edition》。

我永远不会忘记面对的一切，但是对于这种情况，我希望出现一次意外。  
——Groucho Marx（电影人）

# 第 12 章

## 执 行

我们必须团结在一起，否则我们就会被一个一个地绞死。  
——本杰明·弗兰克林（1776 年 7 月 4 日）

本章详细说明了在程序执行期间发生的动作。本章围绕 Java 虚拟机以及构成程序的类、接口和对象的生命周期来组织内容。

Java 虚拟机的启动过程是：首先加载指定的类，然后调用该指定类中的 main 方法。第 12.1 节概括了 main 执行中涉及的加载、链接和初始化步骤，以此介绍本章中的概念。后面几节详细说明了加载（12.2 节）、链接（12.3 节）和初始化（12.4 节）。

本章接下来讨论了创建新的类实例的过程的规范（12.5 节）；以及类实例的终结（finalization）（12.6 节）。最后描述了卸载类（12.7 节），以及程序退出时的后续过程（12.8 节）。

### 12.1 虚拟机启动

Java 虚拟机启动执行的方式是：调用某个指定类中的方法 main，向其传递一个参数，它是一个字符串数组。在本规范的示例中，第一个类通常称为 Test。

《The Java™ Virtual Machine, Specification Second Edition》的第 5 章中给出了虚拟机启动的精确语义。这里，我们从 Java 编程语言的角度概述了这个过程。

将初始类指定给 Java 虚拟机的方式超出了本规范的范围，但是在使用命令行的主机环境中，通常把要指定类的完全限定名称作为命令行参数，并且通常把其后作为字符串使用的命令行参数提供为方法 main 的参数。例如，在 UNIX 实现中，以下命令行：

```
java Test reboot Bob Dot Enzo
```

通常会调用类 Test（未命名包中的类）的方法 main，并将一个包含 4 个字符串（"reboot"、"Bob"、"Dot" 和 "Enzo"）的数组传递给它，以此来启动 Java 虚拟机。

作为后面几节中进一步描述的加载、链接和初始化过程的一个示例，我们现在概述一下虚拟机可能采取的用于执行 Test 的步骤。

### 12.1.1 加载类 Test

在起初尝试执行类 Test 的方法 main 时，会发现类 Test 未加载——也就是说，虚拟机当前不包含该类的二进制表示。然后，虚拟机使用类加载器尝试查找这样一个二进制表示。如果这个过程失败，那么就会抛出错误。第 12.2 节中进一步描述了这个加载过程。

### 12.1.2 链接测试：验证、准备、解析（可选）

在加载 Test 之后，在可以调用 main 之前必须对其进行初始化。与所有（类或接口）类型一样，Test 在初始化之前必须链接。链接涉及验证、准备和解析（可选）。第 12.3 节中进一步描述了链接。

验证检查加载的 Test 表示是否是良构的，并且是否有一份正确的符号表。验证还会检查实现 Test 的代码是否遵守 Java 编程语言和 Java 虚拟机的语义要求。如果在验证期间检测到问题，那么就会抛出一个错误。第 12.3.1 节进一步描述了验证。

准备涉及分配由虚拟机内部使用的静态存储区和任何数据结构，如方法表。第 12.3.2 节进一步描述了准备。

解析是检查从 Test 到其他类和接口的符号引用的过程，通过加载提及的其他类和接口并检查引用是正确的来完成这一过程。

解析步骤在初始链接时是可选的。一个实现可能解析来自于极早被链接的类或接口的符号引用，甚至直到递归地解析来自于被进一步引用的类和接口的所有符号引用（这种解析可能导致这些进一步的加载和链接步骤出错）。这种实现选择代表一种极端的情况，类似于 C 语言的简单实现中已执行许多年的一类“静态”链接（在这些实现中，编译过的程序通常被表示为一个“a.out”文件，它包含程序的完全链接的版本，包括指向程序使用的库例程的完全解析的链接。“a.out”文件中包含有这些库例程的副本）。

相反，一种实现可能选择仅当某个符号引用被主动使用时，才对其进行解析；为所有符号引用一致地使用这种策略代表一种“最懒惰的”解析形式。

在这种情况下，如果 Test 具有多个指向另一个类的符号引用，那么就可以在使用（或者也许根本不使用，如果这些引用在程序执行期间从来不会被使用）它们时，同时解析这些引用。

执行解析时惟一的要求是：当程序直接或间接采取的某个动作可能需要指向错误中涉及的类或接口的链接时，必须抛出解析期间检测到的任何错误。使用上面描述的“静态”示例实现选择，如果加载和链接错误涉及类 Test 中提及的类或接口，或者进一步讲，涉及递归引用的类和接口时，那么在程序执行之前就可能发生加载和链接错误。在实现“最懒惰的”解析方案的系统中，仅当主动使用了不正确的符号引用时，才会抛出这些错误。

第 12.3.3 节进一步描述了解析方案过程。

### 12.1.3 初始化 Test：执行初始化语句

在我们的连续示例中，虚拟机仍然尝试执行类 Test 的方法 main。仅当该类已被初始化时（12.4.1 节），才允许这样做。



初始化包含按代码顺序执行类 `Test` 中的任何类变量初始化语句和静态初始化语句。但是，在可以初始化 `Test` 之前，必须初始化它的直接超类，并递归地初始化它的直接超类的直接超类，依此类推。在最简单的情况下，`Test` 具有 `Object` 作为其隐式直接超类；如果类 `Object` 尚未被初始化，那么在初始化 `Test` 之前，必须对其进行初始化。类 `Object` 没有超类，因此递归在此终止。

如果类 `Test` 具有另一个类 `Super` 作为其超类，那么 `Super` 必须在 `Test` 之前初始化。如果还没有加载、验证和准备 `Super`，则还需要做这些工作，并且依赖于实现，可能还需要涉及递归地解析来自于 `Super` 的符号引用，依此类推。

因此，初始化可能引起加载、链接和初始化错误，包括涉及其他类型的这类错误。

第 12.4 节进一步描述了初始化过程。

#### 12.1.4 调用 `Test.main`

最终，在完成了类 `Test` 的初始化（在这期间，可能发生由之引起的其他加载、链接和初始化）之后，就会调用 `Test` 的方法 `main`。

方法 `main` 必须声明为 `public`、`static` 和 `void` 类型。它必须接受一个字符串数组类型的参数。该方法可被声明为：

```
public static void main(String[] args)
```

或

```
public static void main(String... args)
```

## 12.2 加载类和接口

加载是指查找具有特定名称的类或接口类型的二进制形式，这也许是在不工作时通过计算而得到的，但是更常见的方式是：检索以前由编译器计算源代码得到的二进制表示，并从该二进制形式构造一个表示类或接口的 `Class` 对象。

《The Java™ Virtual Machine Specification》的第 5 章中给出了加载的精确语义（当我们在本书中引用该书时，指的都是由 JSR 924 修改过的第二版）。这里，我们将从 Java 编程语言的角度概述这一过程。

类或接口的二进制格式通常是上面引用的《The Java™ Virtual Machine Specification》中描述的 `class` 文件格式，但是其他格式也可以，只要它们满足第 13.1 节中详细说明的要求。类 `ClassLoader` 的 `defineClass` 方法可用于从 `class` 文件格式中的二进制表示来构造 `Class` 对象。

良好工作的类加载器将维持下面这些性质：

- 给定相同的名称，良好的类加载器应该总是返回相同的类对象。
- 如果类加载器 `L1` 将类 `C` 的加载委托给另一个加载器 `L2`，那么对于作为 `C` 的直接超类或直接超接口、或者作为 `C` 中的方法或构造函数的形参类型、或者作为 `C` 中方法的返回类型出现的任何类型 `T`，`L1` 和 `L2` 应该返回相同的类对象。



恶意的类加载器可能违反这些性质。但是，它不会破坏类型系统的安全性，因为 Java 虚拟机会预防这一点。

有关这些问题的进一步讨论，参见《The Java™ Virtual Machine Specification》以及 Sheng Liang 和 Gilad Bracha 在《Proceedings of OOPSLA '98》撰写的论文“Dynamic Class Loading in the Java™ Virtual Machine”（该论文发表于《ACM SIGPLAN Notices》，第 33 卷，第 10 期，1998 年 10 月，第 36~44 页）。Java 编程语言设计的基本原理是：运行时类型系统不会被用该语言编写的代码破坏，甚至不会被另外一些敏感的系统类（如 `ClassLoader` 和 `SecurityManager`）的实现破坏。

### 12.2.1 加载过程

加载过程是由类 `ClassLoader` 及其子类实现的。`ClassLoader` 的不同子类可能实现不同的加载策略。特别地，类加载器可能缓存类和接口的二进制表示，基于预期的使用预先取出它们，或者一起加载一组相关的类。这些动作对于运行的应用程序可能不是完全透明的，例如，如果由于类加载器缓存了类的旧版本，而使得未找到该类最新编译的版本。但是，类加载器的责任是：仅在程序中可能发生加载错误的那一点反映出错的事实，而无需预先取出或成组加载。

如果在类加载期间发生错误，那么将会在（直接或间接地）使用类型的程序中的任何点抛出类 `LinkageError` 的以下子类之一的实例：

- `ClassCircularityError`：类或接口不能被加载，因为它是其自身的超类或超接口（13.4.4 节）。
- `ClassFormatError`：声称指定了请求的编译类或接口的二进制数据是一种不良的风格。
- `NoClassDefFoundError`：相关的类加载器无法找到请求的类或接口的定义。

由于加载涉及分配新的数据结构，所以它在失败时可能会抛出 `OutOfMemoryError`。

## 12.3 链接类和接口

链接是获取类或接口类型的二进制形式，并将其合并进 Java 虚拟机的运行时状态中，使其能够被执行的过程。类或接口类型在被链接前，总是会被加载。

链接过程中涉及三种不同的动作：验证、准备和解析符号引用。《The Java™ Virtual Machine Specification, Second Edition》的第 5 章中给出了链接的精确语义。这里，我们将从 Java 编程语言的角度概述这一过程。

关于链接活动（由于递归、加载）何时发生，本规范允许实现灵活地做出选择，假定会遵守语言的语义，类或接口在初始化之前会得到彻底验证和准备，并且当程序采取的某个动作可能需要指向错误中涉及的类或接口的链接时，将会抛出链接期间检测到的错误。

例如，一个实现可能选择仅当使用符号引用时，才会解析类或接口中的每个符号引用（懒惰或延迟解析），或者在验证类时同时立即解析它们（静态解析）。这意味着在某些实现

中，在初始化类或接口之后，解析过程可能继续进行。

由于链接涉及分配新的数据结构，所以它在失败时可能会抛出 `OutOfMemoryError`。

### 12.3.1 验证二进制表示

验证确保类或接口的二进制表示都是结构正确的。例如，它会检查每条指令都具有一个有效的操作代码；每条分支指令都会分支到另一条指令的起点，而不是分支进该指令的中间；每个方法都会被提供一个结构正确的签名；并且每条指令都会遵守 Java 虚拟机语言的类型准则。

有关验证过程的规范，参见《The Java™ Virtual Machine Specification》以及 J2ME Connected Limited Device Configuration 1.1 版本的规范。

如果在验证期间发生错误，那么将会在程序中引起类被验证的那一点抛出类 `LinkageError` 的以下子类的实例：

- `VerifyError`：类或接口的二进制定义无法传递一组需要的检查，以验证它是否遵守 Java 虚拟机语言的语义，并且它不能违反 Java 虚拟机的完整性（参见 13.4.2 节、13.4.4 节、13.4.9 节和 13.4.17 节，以了解一些示例）。

### 12.3.2 准备类或接口类型

准备涉及为类或接口创建 `static` 字段（类变量和常量），并把这些字段初始化为默认值（4.12.5 节）。这不需要执行任何源代码；`static` 字段的显式初始化语句是作为初始化（12.4 节）（而不是准备）的一部分执行的。

Java 虚拟机的实现可能在准备期间预先计算额外的数据结构，以使类或接口上的后续操作更高效。一种特别有用的数据结构是“方法表”，或者是无需在调用时搜索超类，即允许在类的实例上调用任何方法的其他数据结构。

### 12.3.3 解析符号引用

类或接口的二进制表示可以使用其他类和接口的二进制名称（13.1 节）以符号方式引用其他的类和接口，以及它们的字段、方法和构造函数（13.1 节）。对于字段和方法，这些符号引用包括：字段或方法是其成员的类或接口类型的名称、字段或方法自身的名称，以及合适的类型信息。

在可以使用符号引用之前，必须对其进行解析，其中要检查符号引用是否正确，并且如果某个引用会反复使用的话，则通常用可以更高效地处理的直接引用代替它。

如果在解析期间发生错误，那么将会抛出一个错误。这通常是类 `IncompatibleClassChangeError` 的以下子类之一的实例，但是它也可以是 `IncompatibleClassChangeError` 的另外某个子类的实例，或者甚至是类 `IncompatibleClassChangeError` 自身的实例。该错误可以在直接或间接地使用指向类型的符号引用的程序中的任意点抛出：

- `IllegalAccessError`：遇到了某个符号引用，它指定了字段的使用或赋值、方法

的调用，或者类实例的创建，包含指向它的引用的代码不具有访问权限，这是由于字段或方法被声明为 `private`、`protected` 或默认访问类型（不是 `public`），或者由于类未被声明为 `public` 类型。

例如，如果在另一个类引用了一个经编译的字段之后，将这个起初声明为 `public` 的字段更改为 `private`，则会发生这种错误（13.4.7 节）。

- `InstantiationError`：遇到了某个符号引用，它用在类实例创建表达式中，但是，由于该引用被证明引用了一个接口或者一个 `abstract` 类，因此不能创建一个实例。例如，如果在另一个类引用了正被议论的类并且编译了前一个类之后，将起初不是 `abstract` 的后一个类更改为 `abstract`，则会发生这种错误（13.4.1 节）。

- `NoSuchFieldError`：遇到了某个符号引用，它引用特定类或接口的特定字段，但是，该类或接口不包含具有那个名称的字段。

例如，如果在另一个类引用了某个字段并且编译了这个类之后，将字段声明从类中删除，则会发生这种错误（13.4.8 节）。

- `NoSuchMethodError`：遇到了某个符号引用，它引用特定类或接口的特定方法，但是，该类或接口不包含具有那种签名的方法。

例如，如果在另一个类引用了某个方法并且编译了这个类之后，将方法声明从类中删除，则会发生这种错误（13.4.12 节）。

此外，如果类声明了一个 `native` 方法，但未找到其实现，则会抛出 `UnsatisfiedLinkError`（`LinkageError` 的子类）。如果使用方法，则会发生一个错误，或者在更早的时间，则依赖于虚拟机正在使用哪种解析策略（12.3 节）。

## 12.4 初始化类和接口

类的初始化包含执行其静态初始化语句，以及类中声明的 `static` 字段（类变量）的初始化语句。接口的初始化包含执行接口中声明的字段（常量）的初始化语句。

在初始化一个类之前，必须先初始化其超类，但是不会初始化由该类实现的接口。类似地，在初始化一个接口之前，不会初始化该接口的超接口。

### 12.4.1 初始化何时发生

类的初始化包含执行其静态初始化语句，以及类中声明的静态字段的初始化语句。接口的初始化包含执行接口中声明的字段的初始化语句。

在初始化一个类之前，必须先初始化其超类，但是不必初始化由该类实现的接口。类似地，在初始化一个接口之前，不必初始化该接口的超接口。

在以下情况中的任意一种情况第一次发生之前，将立即初始化类或接口类型 `T`：

- `T` 是一个类，并且创建 `T` 的一个实例。
- `T` 是一个类，并且调用 `T` 声明的一个静态方法。
- 为 `T` 声明的一个静态字段赋值。

- 使用  $T$  声明的一个静态字段，并且该字段不是一个常变量（4.12.4 节）。
- $T$  是一个顶级类，并且执行了一条从词汇上嵌套在  $T$  内的 `assert` 语句（14.10 节）。

调用类 `Class` 及包 `java.lang.reflect` 中的某些反射方法，也会引起类或接口的初始化。在任何其他情况下，都不会对类或接口进行初始化。

这里要表达的意思是：类或接口类型都有一组将其置于一种一致状态的初始化语句，并且这种状态是其他类观察到的第一种状态。静态初始化语句和类变量初始化语句将会按代码顺序执行，并且不能引用其声明出现在使用之后的类中声明的类变量，即使这些类变量在作用域（8.3.2.3 节）中也是如此。这种限制用于在编译时检测大多数循环的或其他不良的初始化。

如第 8.3.2.3 节中的示例所示，初始化代码不受限制的事实允许按如下方式构造这些示例：在类中，在对其初始化表达式进行求值前，当类变量仍然具有它的初始默认值时，也可以观察到它的值，但是这些示例在实际中很少见（也可以为实例变量初始化构造这样的示例：参见第 12.5 节末尾的示例）。在这些初始化语句中，可以利用 Java 语言的全部能力：程序员必须多加小心。这种能力会把额外的负担放到代码生成器上，但是这种负担无论如何都会发生，因为 Java 语言是一种并发语言（12.4.3 节）。

在初始化一个类之前，如果以前没有初始化它的超类，则必须初始化这些超类。

因此，下面的测试程序：

```
class Super {
    static { System.out.print("Super "); }
}
class One {
    static { System.out.print("One "); }
}
class Two extends Super {
    static { System.out.print("Two "); }
}
class Test {
    public static void main(String[] args) {
        One o = null;
        Two t = new Two();
        System.out.println((Object)o == (Object)t);
    }
}
```

输出如下：

```
Super Two false
```

类 `One` 永远不会被初始化，因为它未被主动使用，因此永远不会被链接。仅当初始化了类 `Two` 的超类 `Super` 之后，才会对类 `Two` 进行初始化。

指向类字段的引用将会导致仅实际声明它的类或接口才会被初始化，即使它可能是通过子类、子接口或者实现接口的类的名称引用的也是如此。

下面的测试程序：

```
class Super { static int taxi = 1729; }
```

```

class Sub extends Super {
    static { System.out.print("Sub "); }
}
class Test {
    public static void main(String[] args) {
        System.out.println(Sub.taxi);
    }
}

```

只会输出:

1729

这是由于类 Sub 永远不会被初始化: 指向 Sub.taxi 的引用是指向在类 Super 中实际声明的字段的引用, 并且不会触发类 Sub 的初始化。

接口自身的初始化不会引起它的任何超接口被初始化。

因此, 下面的测试程序:

```

interface I {
    int i = 1, ii = Test.out("ii", 2);
}
interface J extends I {
    int j = Test.out("j", 3), jj = Test.out("jj", 4);
}
interface K extends J {
    int k = Test.out("k", 5);
}
class Test {
    public static void main(String[] args) {
        System.out.println(J.i);
        System.out.println(K.j);
    }
    static int out(String s, int i) {
        System.out.println(s + "=" + i);
        return i;
    }
}

```

产生以下输出:

```

1
j=3
jj=4
3

```

指向 J.i 的引用是一个字段, 它是一个编译时常量; 因此, 它不会引起 I 被初始化。指向 K.j 的引用是指向在接口 J 中实际声明的字段的引用, 它不是一个编译时常量; 这会引起接口 J 的字段被初始化, 但是不会引起其超接口 I 的字段被初始化, 也不会引起接口 K 的字段被初始化。尽管存在名称 k 用于引用接口 J 的字段 j 这一事实,



接口 `K` 还是不会被初始化。

### 12.4.2 详细的初始化过程

由于 Java 编程语言是一种多线程语言，因此类或接口的初始化需要小心进行同步，这是因为某个其他的线程可能同时尝试初始化相同的类或接口。类或接口的初始化可能作为那个类或接口的初始化的一部分而被递归地请求，这也是有可能出现的一种情况；例如，类 `A` 中的变量初始化语句可能调用不相关的类 `B` 的一个方法，类 `B` 反过来又可能调用类 `A` 的方法。Java 虚拟机的实现通过使用下列过程负责处理同步和递归初始化。它假定已验证和准备了 `Class` 对象，并且 `Class` 对象包含一种指示以下 4 种情况之一的状态：

- 验证和准备了这个 `Class` 对象，但是未对其进行初始化。
  - 正在通过某个特定的线程 `T` 对这个 `Class` 对象进行初始化。
  - 这个 `Class` 对象已被完全初始化，并准备好使用。
  - 这个 `Class` 对象处于一种出错的状态中，这或许是由于尝试进行初始化并且失败了。
- 初始化类或接口的过程如下：

(1) 同步（14.19 节）表示要被初始化的类或接口的 `Class` 对象。这涉及等待，直到当前线程可以获得那个对象的锁（17.1 节）。

(2) 如果正在通过某个其他的线程对类或接口进行初始化，那么 `wait`（等待）这个 `Class` 对象（它会暂时释放锁）。当从 `wait` 唤醒当前线程时，重复这个步骤。

(3) 如果正在通过当前线程对类或接口进行初始化，那么这必定是对初始化的递归请求。释放 `Class` 对象上的锁，并正常完成。

(4) 如果类或接口已被初始化，那么无需进一步的动作。释放 `Class` 对象上的锁，并正常完成。

(5) 如果 `Class` 对象处于一种出错的状态中，那么不可能进行初始化。释放 `Class` 对象上的锁，并抛出一个 `NoClassDefFoundError`。

(6) 否则，记录现在正通过当前线程对 `Class` 对象进行初始化这一事实，并释放 `Class` 对象上的锁。

(7) 接下来，如果 `Class` 对象代表一个类而不是一个接口，并且该类的超类还未被初始化，那么将会为超类递归地执行这一整个过程。如果必要，则首先验证和准备超类。如果由于一个抛出的异常而使超类的初始化突然结束，则锁定这个 `Class` 对象，将其标记为出错，通知所有等待的线程，释放锁，并突然结束，然后抛出由初始化超类产生的相同异常。

(8) 接下来，确定是否通过查询其定义的类加载器，为该类型启用断言（14.10 节）。

(9) 接下来，除了接口的 `final` 类变量和字段（其值是首先被初始化的编译时常量）之外（8.3.2.1 节、9.3.1 节、13.4.9 节），按代码顺序执行类的类变量初始化语句和静态初始化语句，或者接口的字段初始化语句，就像它们位于单独一个语句块中一样。

(10) 如果初始化的执行正常完成，那么就会锁定这个 `Class` 对象，将其标记为完全初始化，通知所有等待的线程，释放锁，并正常完成这个过程。

(11) 否则，必须通过抛出某个异常 `E`，突然完成初始化语句。如果 `E` 的类不是 `Error`



或其子类之一，那么就会创建以 *E* 作为参数的类 `ExceptionInInitializerError` 的一个新实例，并在下一步中使用这个新对象代替 *E*。但是，如果由于发生 `OutOfMemoryError`，而使得不能创建 `ExceptionInInitializerError` 的新实例，则会代之以在下一步中使用 `OutOfMemoryError` 对象代替 *E*。

(12) 锁定 `Class` 对象，将其标记为出错，通知所有等待的线程，释放锁，并由于 *E* 或其替代对象（在上一步中确定）而突然完成这个过程。

（由于某些早期实现中的缺陷，导致会忽视类初始化期间的异常，而不是引发这里描述的 `ExceptionInInitializerError`。）

### 12.4.3 初始化：代码生成的含义

代码生成器需要保持类或接口可能进行初始化的点，插入对刚才描述的初始化过程的调用。如果这个初始化过程正常完成，并且 `Class` 对象完全初始化并准备好使用，那么就不再需要调用初始化过程，并且可能会将其从代码中删除——例如，通过修补代码或重新生成代码。

在某些情况下，如果可以确定一组相关类型的初始化顺序，则编译时分析也许能够消除对通过生成的代码初始化某个类型的许多检查。但是，这样的分析必须充分考虑到并发性，以及初始化代码不受限制的事实。

## 12.5 创建新的类实例

当对类实例创建表达式（15.9 节）进行求值而引起类被实例化时，就会显式创建新的类实例。

在以下情形中，可能隐式创建新的类实例：

- 加载包含 `String` 值（3.10.5 节）的类或接口可能创建一个新的 `String` 对象，用于表示该值 [如果以前已经使用了相同的 `String`（3.10.5 节），则可能不会发生这种情况]。
- 执行某个操作引起装箱转换（5.1.7 节）。装箱转换可能创建与基本类型之一关联的包装器类的一个新对象。
- 执行并非常量表达式一部分的字符串串接运算符（15.18.1 节），有时会创建一个新的 `String` 对象，用于表示结果。字符串串接运算符也可能会为基本类型的值创建临时包装器对象。

这些情形中的每一种情形都利用指定的参数（可能一个也没有）来确定要被调用的特定构造函数，作为类实例创建过程的一部分。

在创建一个新的类实例时，都会为其分配内存空间，该空间适用于类类型中声明的所有实例变量以及该类类型的每个超类中声明的所有实例变量，包括可能被隐藏（8.3 节）的所有实例变量。如果没有足够的空间可用于为对象分配内存，那么类实例的创建就会突然结束，并且会抛出一个 `OutOfMemoryError`。否则，会将新对象中的所有实例变量（包

括那些在超类中声明的实例变量)都初始化成它们的默认值(4.12.5 节)。

刚好在指向新创建的对象引用作为结果被返回之前,将会使用以下步骤处理指定的构造函数,以初始化新对象:

(1) 将构造函数的参数赋予为此构造函数调用新创建的参数变量。

(2) 如果该构造函数开始于对同一个类中的另一个构造函数的显式构造函数调用(使用 `this`),则递归地使用这 5 个相同的步骤对参数求值和处理那个构造函数调用。如果那个构造函数调用突然结束,那么这个过程就会由于相同的原因而突然结束;否则,继续执行第(5)步。

(3) 该构造函数不是开始于对同一个类中的另一个构造函数的显式构造函数调用(使用 `this`)。如果该构造函数针对的是类而不是对象,那么该构造函数将开始于超类构造函数的显式或隐式调用(使用 `super`)。递归地使用这 5 个相同的步骤对参数求值并处理那个构造函数调用。如果那个构造函数调用突然结束,那么这个过程就会由于相同的原因而突然结束;否则,继续执行第(4)步。

(4) 为该类执行实例初始化语句和实例变量初始化语句,依据实例变量出现在类的源代码中的位置,按从左到右的顺序把实例变量初始化语句的值赋予相应的实例变量。如果执行这些初始化语句中的任何一条都会导致一个异常,那么将不会处理更多的初始化语句,并且这个过程会突然结束,同时会抛出那个相同的异常。否则,继续执行第(5)步(在某些早期的实现中,如果字段初始化语句表达式是一个常量表达式,并且其值等于其类型的默认初始化值,那么编译器会不正确地忽略初始化字段的代码)。

(5) 执行该构造函数体的余下部分。如果这个执行突然完成,那么这个过程就会由于相同的原因而突然结束;否则,这个过程会正常结束。

在下面的示例中:

```
class Point {
    int x, y;
    Point() { x = 1; y = 1; }
}
class ColoredPoint extends Point {
    int color = 0xFF00FF;
}
class Test {
    public static void main(String[] args) {
        ColoredPoint cp = new ColoredPoint();
        System.out.println(cp.color);
    }
}
```

会创建 `ColoredPoint` 的一个新实例。首先,为新的 `ColoredPoint` 分配空间,以存储字段 `x`、`y` 和 `color`。然后,将所有这些字段初始化成它们的默认值(在这个例子中,每个字段都为 0)。接下来,将先调用无参的 `ColoredPoint` 构造函数。由于 `ColoredPoint` 没有声明构造函数,Java 编译器会自动为其提供如下形式的默认构造函数:

```
ColoredPoint() { super(); }
```

该构造函数然后调用无参的 Point 构造函数。此 Point 构造函数不会开始于一个构造函数的调用，因此编译器会提供一个对其无参超类构造函数的隐式调用，就像它被写成如下形式一样：

```
Point() { super(); x = 1; y = 1; }
```

因此，将会调用 Object 的无参构造函数。

类 Object 没有超类，因此递归会终止于此。接下来，会调用 Object 的任何实例初始化语句以及实例变量初始化语句。接着会执行 Object 无参的构造函数体。Object 中未声明这样一个构造函数，因此编译器会提供一个默认构造函数，在这个特例中，其形式如下：

```
Object() { }
```

执行该构造函数不会产生任何影响，然后返回。

接下来，将会执行类 Point 的实例变量的所有初始化语句。当这个过程发生时，x 和 y 的声明不会提供任何初始化表达式，因此不需要为本示例的这个步骤采取任何动作。然后执行 Point 构造函数体，将 x 和 y 都设置为 1。

接下来，执行类 ColoredPoint 的实例变量的初始化语句。这个步骤将会把值 0xFF00FF 赋予 color。最后，执行 ColoredPoint 构造函数体的余下部分（super 调用之后的部分）：碰巧在该构造函数体的余下部分中没有任何语句，因此，不需要任何进一步的动作，初始化完成。

与 C++ 不同的是，在创建新的类实例期间，Java 编程语言不会为方法分派来指定变更的规则。如果调用的方法在正被初始化的对象的子类中被重写，那么就会使用这些重写的方法，甚至在新对象被完全初始化之前也是如此。因此，编译和运行这个示例：

```
class Super {
    Super() { printThree(); }
    void printThree() { System.out.println("three"); }
}
class Test extends Super {
    int three = (int) Math.PI;    // That is, 3
    public static void main(String[] args) {
        Test t = new Test();
        t.printThree();
    }
    void printThree() { System.out.println(three); }
}
```

产生如下输出：

```
0
3
```

这表明调用类 Super 的构造函数中的 printThree 不会调用类 Super 中的 printThree 的定义，而会调用类 Test 中 printThree 的重写定义。因此，会在执行 Test 的字段初始化语句之间运行这个方法，这就是为什么第一个值输出结果为 0 的原因，Test 的字段 three 的默认值将被初始化为这个值。在方法 main 中对 printThree 的后一个调用

将会调用 `printThree` 的相同定义，但是，此时已经执行了实例变量 `three` 的初始化语句，因此会打印值 3。

有关构造函数声明的更多信息，参见第 8.8 节。

## 12.6 类实例的终结

类 `Object` 具有一个称为 `finalize` 的 `protected` 方法；该方法可以被其他类重写。可以为某个对象调用的 `finalize` 的特殊定义被称为那个对象的终结器。在垃圾收集器回收为某个对象分配的存储空间之前，Java 虚拟机将调用那个对象的终结器。

对于那些不能通过自动存储管理器自动释放的资源，终结器为释放这些资源提供了一个机会。在这样的情形下，只是简单地回收被对象使用的内存不会保证它占有的资源将会被回收。

在重用对象占有的存储空间之前，Java 编程语言除了指出将调用终结器之外，并没有指定将多久调用一次终结器。此外，该语言也没有指定哪个线程将调用针对任何给定对象的终结器。但是，当调用终结器时，调用终结器的线程将不会占有任何用户可见的同步锁，这一点是得到保证的。如果在终结期间抛出一个未被捕获的异常，则会忽略这个异常，并终止那个对象的终结过程。

对象构造函数的完成“早于 (happen-before)”(17.4.5 节) 其 `finalize` 方法的执行 (就“早于”的正式意义而言)。

### 讨论

许多终结器线程可能是活动的 (在大量共享内存的多处理器上有时需要这样)，如果大量连接的数据结构变成垃圾，则可能同时调用那个数据结构中针对每个对象的所有 `finalize` 方法，并且每个终结器调用运行在一个不同的线程中，注意到这一点是重要的。

类 `Object` 中声明的 `finalize` 方法不会采取任何动作。类 `Object` 声明一个 `finalize` 方法的事实意味着，针对任何类的 `finalize` 方法总是可以调用针对其超类的 `finalize` 方法。这应该始终如此，除非程序员希望使超类中的终结器的动作无效 (与构造函数不同的是，终结器不会自动调用针对超类的终结器；这样一个调用必须显式进行编码)。

出于效率的考虑，一个实现可以跟踪那些没有重写类 `Object` 的 `finalize` 方法或者以无关紧要的方式重写它的类，例如：

```
protected void finalize() throws Throwable {  
    super.finalize();  
}
```

我们鼓励实现把这样的对象视作拥有未被重写的终结器，并且更高效地终结它们，如第 12.6.1 节中所述。

与其他任何方法一样，可以显式调用终结器。

包 `java.lang.ref` 描述了弱引用，它与垃圾收集和终结交互作用。如同任何 API (它

们具有特殊的与 Java 语言的交互作用)一样,实现者必须认识到 java.lang.ref API 强加的任何要求。本规范没有以任何方式讨论弱引用。读者可以参阅 API 文档,以了解详细信息。

### 12.6.1 实现终结

每个对象都可以通过两个属性来表征:它可能是可及的(*reachable*)、终结器可及的(*finalizerreachable*)或不可及的(*unreachable*),也可能是未终结的(*unfinalized*)、可终结的(*finalizable*)或已终结的(*finalized*)。

可及的对象是指可以从任何活着的线程于任何潜在的继续计算中访问的任何对象。可以设计最优化的程序转换,将可及的对象数量减少到少于那些被无经验的新手认为可及的对象数量。例如,编译器或代码生成器可能选择把不再使用的变量或参数设置成 null,以使这样一个对象的存储空间在不久之后潜在地是可回收的。

#### 讨论

如果把对象的字段中的值存储在寄存器中,则会发生这种情况的另一个示例。然后,程序可能访问寄存器而不是对象,并且永远不会再次访问对象。这意味着对象是垃圾。

注意,仅当引用位于栈中而不是堆中时,才允许进行这种优化。

例如,考虑终结器守卫者(*Finalizer Guardian*)模式:

```
class Foo {
    private final Object finalizerGuardian = new Object() {
        protected void finalize() throws Throwable {
            /* finalize outer Foo object */
        }
    }
}
```

如果某个子类重写了终结器,并且没有显式调用 `super.finalize`,则终结器守卫者会强制调用 `super.finalize`。

如果允许对存储在堆上的引用进行这些优化,那么编译器就可以检测到 `finalizer`、`Guardian` 字段永远不会被读取,使之无效,立即收集对象,并及早调用终结器。这运行计数器的目的是:当 `Foo` 实例变量不可及时,程序员可能希望调用 `Foo` 终结器。因此,这种转换不是合法的:只要外部类对象是可及的,那么内部类对象也应该是可及的。

这种转换可能导致 `finalize` 方法的调用比可能预期的时间发生得更早一些。为了允许用户防止这种情况出现,我们加强了同步可以保持对象活着的理念。如果某个对象的终结器可以导致该对象上的同步,那么这个对象必定活着,并且无论何时它拥有一个锁,都将该对象视为是可及的。

注意,这不会阻止同步消除:如果终结器可能同步某个对象,则同步只会保持该对象活着。由于终结器出现在另一个线程中,因此在许多情况下,无论如何也不能删除同步。

对于终结器可及的对象,可以通过某条引用链从某个可终结的对象访问它,但是不能从任何活着的线程访问它。通过任何一种手段都不能访问不可及的对象。



未终结的对象永远不会使其终结器被自动调用；已终结的对象会使其终结器被自动调用。可终结的对象永远不会使其终结器被自动调用，但是 Java 虚拟机最终可能自动调用其终结器。

直到对象  $o$  的构造函数在  $o$  上调用了针对 `Object` 的构造函数，并且该调用成功完成（也就是说，没有抛出异常）时，对象  $o$  才是可终结的。写到某个对象的字段的每个预终结对于那个对象的终结必须是可见的。此外，那个对象的字段的任何预终结读动作都不会看到在启动那个对象的终结之后发生的写动作。

#### 12.6.1.1 与内存模型交互

内存模型（第 17 章）必须能够决定它何时能够提交终结器中发生的动作。本节描述了终结与内存模型的交互。

每个执行都有许多可及性决定点，标记为  $di$ 。每个动作要么早于（*come-before*） $di$ ，要么晚于（*come-after*） $di$ 。除了显式提及的之外，本节中描述的“早于”次序与内存模型中的所有其他次序无关。

如果  $r$  是看到一个写动作  $w$  的读动作，并且  $r$  “早于”  $di$ ，那么  $w$  必定“早于”  $di$ 。如果  $x$  和  $y$  是同一个变量或监视器上的同步动作，满足  $so(x, y)$ （17.4.4 节），且  $y$  “早于”  $di$ ，那么  $x$  必定“早于”  $di$ 。

在每个可及性决定点，将对象的某个集合标记为不可及的，并将那些对象的某个子集标记为可终结的。这些可及性决定点也是检查、排队和清除引用的点，依据针对包 `java.lang.ref` 的 API 文档中提供的规则对引用执行这些操作。

只有那些可以被下面这些规则的应用证明是可及的对象，才会在点  $di$  处被视为一定是可及的：

- 如果存在对类  $C$  的一个静态字段  $v$  的写动作  $w1$ ，满足  $w1$  写的值是一个指向  $B$  的引用，类  $C$  是通过可及的类加载器加载的；并且不存在一个对  $v$  的写动作  $w2$ ，满足  $hb(w2, w1)$  不为真，并且  $w1$  和  $w2$  都“早于”  $di$ ，那么对象  $B$  在  $di$  处对于静态字段必定是可及的。
- 如果存在一个对  $A$  的某个元素  $v$  的写动作  $w1$ ，满足  $w1$  写的值是一个指向  $B$  的引用；并且不存在一个对  $v$  的写动作  $w2$ ，满足  $hb(w2, w1)$  不为真，并且  $w1$  和  $w2$  都“早于”  $di$ ，那么对象  $B$  在  $di$  处对于  $A$  必定是可及的。
- 如果对象  $C$  对于对象  $B$  必定是可及的，并且对象  $B$  对于对象  $A$  必定是可及的，那么  $C$  对于  $A$  也必定是可及的。

当且仅当下列条件中至少有一个成立时，动作  $a$  才会主动使用  $X$ ：

- $a$  读或写  $X$  的一个元素。
- $a$  锁定  $X$  或解除对它的锁定，并且在调用针对  $X$  的终结器之后，在  $X$  上发生了一个锁定动作。
- $a$  写入一个指向  $X$  的引用。
- $a$  主动使用对象  $Y$ ，并且  $X$  对于  $Y$  必定是可及的。

如果对象  $X$  在  $di$  处被标记为不可及的，那么：



- $X$  在  $di$  处对于静态字段绝对不是可及的。
- “晚于”  $di$  的线程  $t$  中对  $X$  的主动使用必须发生在针对  $X$  的终结器调用中，或者作为线程  $t$  执行一个读动作（它“晚于”一个指向  $X$  的引用的  $di$ ）的结果。
- “晚于”  $di$  的看到一个指向  $X$  的引用的所有读动作，都必须看到对  $di$  处不可及对象的元素的写动作，或者看到“晚于”  $di$  的写动作。

如果对象  $X$  在  $di$  处被标记为可终结的，那么：

- $X$  在  $di$  处必须被标记为不可及的。
- $di$  必须是将  $X$  标记为可终结的惟一位置。
- 在终结器调用之后发生的动作必须“晚于”  $di$ 。

## 12.6.2 终结器调用是无序的

Java 编程语言没有对 `finalize` 方法调用的顺序强加任何限制。终结器可以以任何顺序调用，或者甚至并发调用。

例如，如果一组循环链接的未终结对象变成不可及的（或者终结器可及的），那么所有对象都可能一起变成可终结的。最终，可能以任何顺序调用针对这些对象的终结器，或者甚至使用多个线程并发调用它们。如果自动存储管理器后来发现对象是不可及的，那么就可以回收它们的存储空间。

当所有对象变成不可及的时，可以实现一个类，它将导致为这组对象以指定的顺序调用一组类似于终结器的方法，这个类的实现是简单、直观的。把这样一个类的定义留作读者的练习。

## 12.7 卸载类和接口

Java 编程语言的实现可以卸载类。当且仅当类或接口定义的类加载器可以被垃圾收集器回收时（如第 12.6 节中的讨论），才可以卸载类或接口。通过引导程序加载器加载的类和接口不能被卸载。

下面是上一段中给出的规则的基本原理：

类卸载是一种优化，它有助于减少内存使用。显然，程序的语义不应该依赖于系统是否以及如何选择实现一种优化（如类卸载）。另一方面，这样做将会折衷程序的可移植性。因此，是否卸载了类或接口对于程序应该是透明的。

但是，如果卸载了类或接口  $C$ ，同时其定义的加载器是潜在可及的，那么就可能重新加载  $C$ 。人们永远不能确保这不会发生。即使一个类未被当前加载的其他任何类引用，它也可能被某个尚未加载的类或接口  $D$  引用。当  $D$  通过  $C$  定义的加载器加载时，它的执行将引起  $C$  被重新加载。

例如，如果类具有以下成员，则重新加载可能不是透明的：

- 静态变量（其状态将被丢失）。
- 静态初始化语句（它可能具有副作用）。

- 本地方法（它可能保持静态状态）。

此外，Class 对象的散列值依赖于其身份。因此，一般来讲，不可能以一种完全透明的方式重新加载类或接口。

既然我们永远不能保证卸载一个类或接口（其加载器是潜在可及的）将不会引起重新加载，并且重新加载永远不会是透明的，但是卸载必须是透明的，因此人们必须遵循一条规则：当类或接口的加载器是潜在可及的时，绝对禁止卸载类或接口。可以使用类似的推理准线推断出：永远不能卸载通过引导程序加载器加载的类和接口。

如果类 C 定义类加载器可以被回收，那么还必须证明为什么卸载类 C 是安全的。如果定义类加载器可以被回收，那么永远不能有一个指向它的活着的引用（这包括那些不是活着的，但是可以通过终结器使之复活的引用）。这样，仅当永远不会有任何活着的引用通过类的实例或者代码指向由那个加载器定义的任何类（包括 C）时，这种情况才会成立。

类卸载是一种优化，它只对于加载大量类以及在一段时间后停止使用其中大部分类的应用才是重要的。这样一个应用的首要示例是 Web 浏览器，但是也有一些其他的应用。这些应用的特征是：它们通过显式使用类加载器来管理类。因此，上面概括的策略很适合它们。

严格地讲，本规范没有必要讨论类卸载的问题，因为类卸载只是一种优化。但是，这个问题非常深奥，因此这里通过阐述的方式提及了它。

## 12.8 程序退出

当下面两件事情之一发生时，程序就会终止其所有的活动并退出：

- 所有不是守护线程的线程都终止。
- 某个线程调用类 Runtime 或类 System 的 exit 方法，并且该退出操作没有被安全管理器禁止。

别了！天色越变越坏，你多半要听到一阕太粗暴的催眠歌。  
我从不曾见过白昼的天色会这么阴暗。哪里来的怕人的喧声！

但愿我平安上了船！一头野兽给人赶到这儿来了；

我这回准活不成！（被熊追下。）

——威廉·莎士比亚，《冬天的故事》第三幕第三场

## 二进制兼容性

不管面向对象编程中的软件重用有怎样的许诺，这种技术实际上已经用尽了全部潜能。重用的主要障碍在于，无法在继续支持已编译应用程序的情况下发展编译类库……必须谨慎设计面向对象的模型，以使类库转换不会破坏已编译的应用程序，实际上，是不能破坏此类应用程序。

——Ira Forman、Michael Conner、Scott Danforth 和 Larry Raper,  
《Release-to-Release Binary Compatibility in SOM》(1995)

Java 编程语言的开发工具应该支持：无论何时源代码可用，都可根据需要进行自动重新编译。特定的实现还可以把类型的源代码和二进制代码存储在版本化的数据库中，并实现一个 `ClassLoader`，它使用了数据库的完整性机制，通过把类型的二进制兼容的版本提供给客户，来防止链接错误。

将被广泛分发的包和类的开发人员面临一组不同的问题。在 Internet 中，其示例是我们偏爱的广泛分布的系统，它通常是不实用的，或者不可能自动重新编译预先存在的二进制代码，它直接或间接依赖于将被更改的类型。作为替代，本规范定义了允许开发人员对包或者类或接口类型所做的一组更改，同时保持（而非中断）与现有二进制代码的兼容性。

上面引用的论文出现在《Proceedings of OOPSLA '95》中，它收录于《ACM SIGPLAN Notices》第 30 卷第 10 册，1995 年 10 月，第 426~438 页。在该论文的框架内，Java 编程语言的二进制代码在作者确定的所有相关转换下都是二进制兼容的（具有一些关于增加实例变量的告诫）。使用他们的模式，下面列出了 Java 编程语言支持的一些重要的二进制兼容的改变：

- 重新实现现有的方法、构造函数和初始化语句，以改进性能。
- 更改方法或构造函数，以返回关于输入的值，对于这些输入，它们以前抛出过异常，这在正常情况下不应该发生，或者由于进入无限循环或引起死锁而失败。
- 添加新的字段、方法或构造函数到现有的类或接口中。
- 删除类的 `private` 字段、方法或构造函数。
- 在更新整个包时，删除包中的类和接口的默认（仅针对包）访问字段、方法或构造函数。

- 在现有的类型声明中重新排列字段、方法或构造函数的顺序。
- 在类层次结构中向上移动方法。
- 重新排列类或接口的直接超接口的列表。
- 在类型层次结构中插入新的类或接口类型。

本章详细说明了由所有实现保证的二进制兼容性的最低标准。当混合不能通过兼容的源代码得知的类和接口的二进制代码时，Java 编程语言会保证兼容性，但是将会以这里描述的兼容方式对其源代码进行修改。注意，我们正在讨论应用程序版本之间的兼容性。有关 Java 平台版本之间的兼容性的讨论超出了本章的范围。

我们鼓励开发系统提供一些工具，以提醒开发人员有关更改也许不能重新编译的预先存在的二进制代码的影响。

本章首先详细说明了 Java 编程语言必须具有的任何二进制格式的一些性质（13.1 节）；接下来定义了二进制兼容性，解释了它是什么以及它不是什么（13.2 节）；最后枚举了一大组可能对包（13.3 节）、类（13.4 节）和接口（13.5 节）所做的更改，详细说明了这些更改中有哪些会保证保持二进制兼容性，而哪些又不会。

## 13.1 二进制的形式

程序必须被编译进《The Java™ Virtual Machine Specification》指定的 class 文件格式中，或者编译进一种表示中，该表示可通过 Java 编程语言编写的类加载器映射进那种文件格式中。此外，得到的 class 文件必须具有某些属性。其中许多属性是特别选择用于支持保持二进制兼容性的源代码转换。

需要的属性有：

- 类或接口必须通过其二进制名称来命名，该二进制名称必须满足以下约束条件：
  - ◆ 顶级类型的二进制名称是其规范名称（6.7 节）。
  - ◆ 成员类型的二进制名称包含其直接封闭类型的二进制名称，后接一个\$符号，再接该成员的简单名称。
  - ◆ 本地类（14.3 节）的二进制名称包含其直接封闭类型的二进制名称，后接一个\$符号，然后接一个非空数字序列，再接该本地类的简单名称。
  - ◆ 匿名类（15.9.5 节）的二进制名称包含其直接封闭类型的二进制名称，后接一个\$符号，再接一个非空数字序列。
  - ◆ 由泛型类或接口声明的类型变量的二进制名称包含其直接封闭类型的二进制名称，后接一个\$符号，再接该类型变量的简单名称。
  - ◆ 由泛型方法声明的类型变量的二进制名称包含声明该方法的类型的二进制名称，后接一个\$符号，然后接《The Java™ Virtual Machine Specification》中定义的方法的描述符，再接一个\$符号，最后接该类型变量的简单名称。
  - ◆ 由泛型构造函数声明的类型变量的二进制名称包含声明该构造函数的类型的二进制名称，后接一个\$符号，然后接《The Java™ Virtual Machine Specification》中定义的构造函数的描述符，再接一个\$符号，最后接该类型变量的简单名称。



- 指向另一个类或接口类型的引用必须是符号引用，并使用该类型的二进制名称。
- 给定一个指示类 *C* 中字段访问的合法表达式，引用一个（可能不同的）类或接口 *D* 中声明的名为 *f* 的非常量（13.4.9 节）字段，我们把字段引用的合格类型定义如下：
  - ◆ 如果表达式形如 *Primary.f*，那么：
    - ◇ 若 *Primary* 的编译时类型是一个交集类型（4.9 节）*V1 & ... & Vn*，那么该引用的合格类型就是 *V1*。
    - ◇ 否则，*Primary* 的编译时类型就是该引用的合格类型。
  - ◆ 如果表达式形如 *super.f*，那么 *C* 的超类就是该引用的合格类型。
  - ◆ 如果表达式形如 *X.super.f*，那么 *X* 的超类就是该引用的合格类型。
  - ◆ 如果该引用形如 *X.f*，其中 *X* 表示类或接口，那么由 *X* 表示的类或接口就是该引用的合格类型。
  - ◆ 如果通过简单名称引用表达式，那么如果 *f* 是当前类或接口 *C* 的成员，则令 *T* 为 *C*。否则，令 *T* 为代码中最内层的封闭类，其中 *f* 是它的一个成员。*T* 就是该引用的合格类型。

必须将指向 *f* 的引用编译成一个符号引用，该符号引用指向那个引用的合格类型的擦除（4.6 节），以及字段 *f* 的简单名称。该引用还必须包括一个符号引用，该符号引用指向字段的声明类型的擦除，以使验证者可以检查类型正是所想要的。

- 如果字段是常变量（4.12.4 节），则会在编译时将指向字段的引用解析成它所表示的常量值。二进制文件的代码中不应该存在指向这样一个常量字段的引用（除非在包含常量字段的类或接口中，其中将具有初始化它的代码），并且这种常量字段必须总是看起来经过了初始化；这种字段的类型的默认初始值决不会被观察到。有关的讨论请参见第 13.4.8 节。
- 在类或接口 *C* 中给定一个方法调用表达式，并且 *C* 引用了（可能不同的）类或接口 *D* 中声明的名为 *m* 的方法，我们把方法调用的合格类型定义如下：

如果 *D* 是 *Object*，那么表达式的合格类型就是 *Object*。否则：

- ◆ 如果表达式形如 *Primary.m*，那么：
  - ◇ 若 *Primary* 的编译时类型是一个交集类型（4.9 节）*V1 & ... & Vn*，那么该方法调用的合格类型就是 *V1*。
  - ◇ 否则，*Primary* 的编译时类型就是该方法调用的合格类型。
- ◆ 如果表达式形如 *super.m*，那么 *C* 的超类就是该方法调用的合格类型。
- ◆ 如果表达式形如 *X.super.m*，那么 *X* 的超类就是该方法调用的合格类型。
- ◆ 如果该引用形如 *X.m*，其中 *X* 表示类或接口，那么由 *X* 表示的类或接口就是该方法调用的合格类型。
- ◆ 如果通过简单名称引用表达式，那么如果 *m* 是当前类或接口 *C* 的成员，则令 *T* 为 *C*。否则，令 *T* 为代码中最内层的封闭类，其中 *m* 是它的一个成员。*T* 就是该引用的合格类型。

必须在编译时将指向一个方法的引用解析成一个符号引用，该符号引用指向那个调用的合格类型的擦除（4.6 节），以及方法签名（8.4.2 节）的擦除。指向一个方法的

引用还必须包括一个符号引用，该符号引用所指示方法的返回类型的擦除；或者包括一个指示器，它指定所指示方法被声明为 `void` 并且不会返回值。方法的签名必须包括以下部分：

- ◆ 方法的简单名称。
- ◆ 方法参数的成员。
- ◆ 指向每一个参数类型的符号引用。
- 在类或接口 `C` 中给定一个类实例创建表达式（15.9 节）或者一条构造函数调用语句（8.8.7.1 节），并且 `C` 引用了（可能不同的）类或接口 `D` 中声明的名为 `m` 的构造函数，我们把构造函数调用的合格类型定义如下：
  - ◆ 如果表达式形如 `new D(...)` 或 `X.new D(...)`，那么该调用的合格类型就是 `D`。
  - ◆ 如果表达式形如 `new D(..){...}` 或 `X.new D(..){...}`，那么表达式的合格类型就是该表达式的编译时类型。
  - ◆ 如果表达式形如 `super(...)` 或 `Primary.super(...)`，那么该表达式的合格类型就是 `C` 的直接超类。
  - ◆ 如果表达式形如 `this(...)`，那么该表达式的合格类型就是 `C`。

必须在编译时将指向一个构造函数的引用解析成一个符号引用，该符号引用指向那个调用的合格类型的擦除（4.6 节），以及构造函数的签名（8.8.2 节）。构造函数的签名必须包括以下两部分：

- ◆ 构造函数的参数数量。
- ◆ 指向每个参数类型的符号引用。

此外，还必须编译非私有内部成员类的构造函数，以使它具有一个额外的隐式参数作为它的第一个参数，该参数用于表示直接封闭的实例（8.1.3 节）。

- 除了默认构造函数和类初始化方法之外，对于编译器引入的任何构造，如果在源代码中没有与之对应的构造，那么必须将其标记为“人造的”。

类或接口的二进制表示还必须包含以下所有部分：

- 如果它是一个类，但不是类 `Object`，那么就要包含一个指向该类直接超类的擦除的符号引用。
- 一个指向每个直接超接口（如果有的话）的擦除的符号引用。
- 类或接口中声明的每个字段的规范，它是作为字段的简单名称以及指向字段类型的擦除的符号引用给出的。
- 如果它是一个类，那么就要包含每个构造函数的擦除签名，如上所述。
- 对于类或接口中声明的每个方法，要包含其擦除签名和返回类型，如上所述。
- 实现类或接口所需要的代码：
  - ◆ 对于接口，要包含字段初始化语句的代码。
  - ◆ 对于类，要包含字段初始化语句、实例和静态初始化语句以及每个方法或构造函数实现的代码。
- 每种类型都必须包含足够的信息，以恢复其规范名称（6.7 节）。实现类或接口所需要的代码。



- 每种成员类型都必须具有足够的信息，以恢复其源级别访问修饰符。
- 每个嵌套类都必须具有一个指向其直接封闭类的符号引用。
- 每个包含嵌套类的类都必须包含指向其所有成员类的符号引用，以及指向出现在其方法、构造函数和静态或实例初始化语句中的所有本地类和匿名类的引用。

下面几节讨论了在不中断与预先存在的二进制代码的兼容性的情况下，可以对类和接口类型声明所做的更改。在上面给出的转换要求之下，Java 虚拟机及其 class 文件格式支持这些更改。在上述要求之下，通过类加载器映射回 class 文件中的其他任何有效的二进制格式（如压缩或加密表示）也必须支持这些更改。

## 13.2 二进制兼容性是什么，不是什么

如果以前无错链接的预先存在的二进制代码将继续无错链接，那么对某个类型所做的更改与预先存在的二进制代码就是二进制兼容的（换句话说，不会中断二进制兼容性）。

二进制代码的编译依赖于其他类和接口的可访问的成员和构造函数。为了保持二进制兼容性，类或接口应该将其可访问的成员和构造函数以及它们的存在状况和行为视作与其用户的契约。

Java 编程语言被设计成防止由于向契约中增加内容以及意外的名称冲突而中断二进制兼容性；确切地讲：

- 增加多个重载特定方法名称的方法不会中断与预先存在的二进制代码的兼容性。预先存在的二进制代码将用于方法查找的方法签名是在编译时通过方法重载解析算法选择的（15.12.2 节）（如果语言被设计成在运行时选择要执行的特定方法，那么可能在运行时检测到这种歧义。这种规则暗示：增加额外的重载方法有可能导致在某个调用位置出现歧义，从而可能中断与未知数量的预先存在的二进制代码的兼容性。有关更多讨论，参见第 13.4.23 节）。

二进制兼容性不同于源代码兼容性。特别地，第 13.4.6 节中的示例显示：可以从不在一起编译的源代码产生一组兼容的二进制代码。这个示例是典型的：添加了新的声明，更改了源代码中未改变部分中的名称的含义，同时源代码中未改变部分的预先存在的二进制代码保持完全限定的、先前的名称含义。产生一组一致的源代码需要提供与先前含义对应的限定名称或字段访问表达式。

## 13.3 包的演变

可以在不中断与预先存在的二进制代码的兼容性的情况下，将新的顶级类或接口类型添加到包中，假定新类型没有重用以前赋予一种无关类型的名称。如果新类型重用了以前赋予一种无关类型的名称，那么就有可能发生冲突，这是由于两种类型的二进制代码不能被同一个类加载器加载。

如果顶级类和接口不是 public 类型，或者它们分别不是 public 类型的超类或超接口，那么对它们所做的更改将只会影响声明它们的包内的类型。这些类型可能被删除，或者被更

改，即使存在这里的不兼容性也是如此，假定那个包中受影响的二进制代码将会一起更新。

## 13.4 类的演变

本节描述了在预先存在的二进制代码上更改类及其成员和构造函数的声明所起的作用。

### 13.4.1 abstract 类

如果把一个非 `abstract` 类更改成声明了的 `abstract`，那么试图创建那个类的新实例的预先存在的二进制代码将会在链接时抛出一个 `InstantiationError`，或者（如果使用了反射方法）在运行时抛出一个 `InstantiationException`；因此，不建议对广泛分发的类执行这种更改。

将声明为 `abstract` 的类更改成不再被声明为 `abstract`，这种更改不会中断与预先存在的二进制代码的兼容性。

### 13.4.2 final 类

如果把一个未声明为 `final` 的类更改成声明了的 `final`，那么如果加载这个类的预先存在的子类的二进制代码，则会抛出一个 `VerifyError`，因为 `final` 类不能有任何子类；不建议对广泛分发的类执行这种更改。

将声明为 `final` 的类更改成不再被声明为 `final`，这种更改不会中断与预先存在的二进制代码的兼容性。

### 13.4.3 public 类

把一个未声明为 `public` 的类更改成声明了的 `public`，这种更改不会中断与预先存在的二进制代码的兼容性。

如果把一个声明为 `public` 的类更改成不将其声明为 `public`，那么如果预先存在的二进制代码在链接时需要访问该类类型，但是它不再具有这种访问权限，则会抛出一个 `IllegalAccessError`；不建议对广泛分发的类执行这种更改。

### 13.4.4 超类和超接口

如果某个类是其自身的超类，则会在加载时抛出一个 `ClassCircularityError`。当把最近编译的二进制代码与预先存在的二进制代码一起加载时，更改类层次结构就可能导致这种循环性，不建议对广泛分发的类执行这种更改。

更改某个类类型的直接超类或直接超接口集合，不会中断与预先存在的二进制代码的兼容性，假定该类类型的超类或超接口的全部集合不会丢失成员。

如果对直接超类或直接超接口的集合所做的更改导致任何类或接口分别不再是超类或超接口，那么如果把预先存在的二进制代码与修改过的类的二进制代码一起加载，则可能

会导致链接时错误。不建议对广泛分发的类执行这种更改。

例如，假定编译并执行下列测试程序：

```
class Hyper { char h = 'h'; }
class Super extends Hyper { char s = 's'; }
class Test extends Super {
    public static void printH(Hyper h) {
        System.out.println(h.h);
    }
    public static void main(String[] args) {
        printH(new Super());
    }
}
```

产生如下输出：

h

假定然后编译类 Super 的新版本：

```
class Super { char s = 's'; }
```

类 Super 的这个版本不是 Hyper 的子类。如果我们随后用 Super 的新版本运行 Hyper 和 Test 现有的二进制代码，则会在链接时抛出一个 `VerifyError`。由于不能把 `new Super()` 的结果作为参数传递，因此用验证器对象代替类型 Hyper 的形参，因为 Super 不是 Hyper 的子类。

考虑在不执行验证步骤的情况下会发生什么是有意义的：程序可能运行并输出：

s

这证明在没有验证器的情况下，链接不一致的二进制文件不会破坏类型系统，即使每个二进制文件是由正确的 Java 编译器产生的也是如此。

一个教训是：缺少验证器或无法使用验证器的实现将不会维持类型安全，因此，不是一个有效的实现。

### 13.4.5 类的形式类型参数

重命名一个被声明为类的形式类型参数的类型变量（4.4 节），不会对预先存在的二进制代码产生任何影响。添加或删除类型参数，本质上不会牵涉到二进制兼容性。



注意，如果在字段或方法的类型中使用这种类型变量，则通常可能牵涉到更改上述类型。

更改类型参数的第一次绑定将会更改在它自己的类型中使用那个类型变量的任何成员的擦除（4.6 节），并且这可能会影响二进制兼容性。更改任何其他的绑定都不会对二进制兼容性产生任何影响。

### 13.4.6 类体和成员声明

添加具有相同名称和可访问性（对于字段）或者具有相同名称、可访问性、签名和返回类型（对于方法）的实例（或 static）成员作为超类或子类的实例（或 static）成员，不会引起与预先存在的二进制代码的不兼容性。即使正被链接的类集合遇到一个编译时错误，也不会发生错误。

如果某个未被声明为 `private` 的类成员或构造函数被预先存在的二进制代码使用，那么删除该类成员或构造函数可能会引起链接错误：

如果编译并执行下列程序：

```
class Hyper {
    void hello() { System.out.println("hello from Hyper"); }
}
class Super extends Hyper {
    void hello() { System.out.println("hello from Super"); }
}
class Test {
    public static void main(String[] args) {
        new Super().hello();
    }
}
```

则会产生如下输出：

```
hello from Super
```

假定创建了类 `Super` 的以下新版本：

```
class Super extends Hyper { }
```

然后重新编译 `Super`，并执行这个具有 `Test` 和 `Hyper` 的原始二进制代码的新二进制文件，将会产生所预期的输出：

```
hello from Hyper
```

`super` 关键字可用于绕过当前类中声明的任何方法，来访问超类中声明的方法。将会在编译时将下面的表达式：

```
super.Identifier
```

解析到超类 `S` 中的方法 `M`。如果方法 `M` 是一个实例方法，那么在运行时调用的方法 `MR` 就是与 `M` 具有相同签名的方法，其中 `M` 是包含某个表达式（该表达式涉及 `super`）的类的直接超类的成员。因此，如果编译并执行下列程序：

```
class Hyper {
    void hello() { System.out.println("hello from Hyper"); }
}
class Super extends Hyper { }
class Test extends Super {
    public static void main(String[] args) {
        new Test().hello();
    }
}
```

```

    }
    void hello() {
        super.hello();
    }
}

```

则会产生如下输出：

```
hello from Hyper
```

假定创建类 Super 的下列新版本：

```

class Super extends Hyper {
    void hello() { System.out.println("hello from Super"); }
}

```

如果重新编译 Super 和 Hyper，但不重新编译 Test，那么运行带有 Test 的现有二进制代码的新二进制文件，将会产生预期的输出：

```
hello from Super
```

某些早期实现中的一个缺陷将会引起它们不正确地输出：

```
hello from Hyper
```

### 13.4.7 访问成员和构造函数

更改成员或构造函数声明的访问方式以允许较少的访问权限，这可能会中断与预先存在的二进制代码的兼容性，从而导致在解析这些二进制代码时抛出链接错误。如果将访问修饰符从默认访问更改为 private 访问；从 protected 访问更改为默认或 private 访问；从 public 访问更改为 protected、默认或 private 访问，则都只会允许较少的访问权限；因此，建议不要对广泛分发的类更改成员或构造函数以允许较少的访问。

也许使人感到奇怪的是，当子类（已经）定义了一个具有较少访问权限的方法时，可以定义一种二进制格式，以使得将成员或构造函数更改成具有更多访问权限不会引起链接错误。

因此，例如，如果包 points 定义了类 Point：

```

package points;
public class Point {
    public int x, y;
    protected void print() {
        System.out.println("(" + x + ", " + y + ")");
    }
}

```

它被 Test 程序使用：

```

class Test extends points.Point {
    protected void print() {
        System.out.println("Test");
    }
    public static void main(String[] args) {
        Test t = new Test();
    }
}

```



```

        t.print();
    }
}

```

然后编译这些类，并执行 Test 产生以下输出：

```
Test
```

如果将类 Point 中的方法 print 更改为 public，然后仅重新编译 Point 类，接着把它与以前存在的 Test 的二进制代码放在一起执行，则不会发生链接错误，即使它在编译时是不正确的也是如此，因为一个 public 方法被一个 protected 方法重写（正如以下事实所示，不能使用这个新的 Point 类重新编译类 Test，除非将输出更改为 public）。

允许超类将 protected 方法更改为 public，而不会中断预先存在子类的二进制代码的兼容性，这有助于使二进制代码变量不那么脆弱。另一种替代方案将会产生额外的二进制不兼容性，在其中执行这种更改将会引发链接错误。

### 13.4.8 字段声明

广泛分发的程序不应该向其客户公开任何字段。除了下面讨论的二进制兼容性问题之外，这通常是一种良好的软件工程实践。添加字段到类中可能中断与不会重新编译的预先存在的二进制代码的兼容性。

假定一个引用指向具有合格类型  $T$  的字段  $f$ 。进一步假定  $f$  事实上是  $T$  的超类  $S$  中声明的一个实例（或 static）字段，并且  $f$  的类型是  $X$ 。如果将具有与  $f$  相同名称的类型  $X$  的新字段添加到  $S$  的子类中，并且该子类是  $T$  的超类或  $T$  本身，那么就可能发生链接错误。除了上述情况外，仅当下列条件之一成立时，才会发生这种链接错误：

- 与旧字段相比，新字段具有较少的访问权限。
- 新字段是 static（或实例）字段。

特别地，如果由于字段访问以前引用了一个具有不兼容类型的超类的字段，而使得不再能够重新编译类，在这种情况下，将不会发生链接错误。以前编译的具有这种引用的类将继续引用超类中声明的字段。

因此，编译并执行下列代码：

```

class Hyper { String h = 'hyper'; }
class Super extends Hyper { String s = 'super'; }
class Test {
    public static void main(String[] args) {
        System.out.println(new Super().h);
    }
}

```

将产生以下输出：

```
hyper
```

将 Super 的定义更改如下：

```

class Super extends Hyper {
    String s = 'super';
}

```



```
int h = 0;
}
```

重新编译 Hyper 和 Super, 并把得到的新二进制代码与 Test 的旧二进制代码放在一起执行, 产生如下输出:

hyper

Hyper 的字段 h 是由 main 原来的二进制代码输出的。虽然这乍看起来可能让人觉得奇怪, 但是, 它可用于减少运行时出现的不兼容性的数量 (在理想世界中, 对于所有需要重新编译的源文件, 只要更改了它们中的任何一个文件, 都将重新编译所有这些文件, 从而消除了这种奇怪的情况。但是大量地进行这种重新编译通常是不切实际的或者是不可能的, 特别是在 Internet 中。此外, 如前所述, 这种重新编译有时需要进一步更改源代码)。

例如, 如果编译并执行下列程序:

```
class Hyper { String h = 'Hyper'; }
class Super extends Hyper { }
class Test extends Super {
    public static void main(String[] args) {
        String s = new Test().h;
        System.out.println(s);
    }
}
```

它将产生如下输出:

Hyper

假定之后重新编译类 Super 的新版本:

```
class Super extends Hyper { char h = 'h'; }
```

如果把得到的二进制代码与 Hyper 和 Test 现有的二进制代码放在一起使用, 那么输出仍然是:

Hyper

即使为这些二进制代码编译源代码:

```
class Hyper { String h = 'Hyper'; }
class Super extends Hyper { char h = 'h'; }
class Test extends Super {
    public static void main(String[] args) {
        String s = new Test().h;
        System.out.println(s);
    }
}
```

也会导致编译时错误, 因为 main 的源代码中的 h 现在将被解释成引用 Super 中声明的 char 字段, 但是不能把一个 char 值赋予一个 String。

从类中删除一个字段将中断与任何预先存在的引用该字段的二进制代码的兼容性, 并且在链接这样一个来自预先存在的二进制代码的引用时, 将会抛出一个 `NoSuchFieldError`。只有 private 字段可以从广泛分发的类中安全地删除。

出于二进制兼容性考虑, 添加或删除其类型涉及变量类型 (4.4 节) 或参数化类型 (4.5 节) 的字段 *f*, 等价于添加 (或删除) 一个相同名称的字段, 并且其类型是 *f* 的类型的擦除 (4.6 节)。

### 13.4.9 final 字段和常量

如果将一个非 final 字段更改为 final, 那么它可能中断与试图将一个新值赋予该字段的预先存在的二进制代码的兼容性。例如, 如果编译并执行以下程序:

```
class Super { static char s; }
class Test extends Super {
    public static void main(String[] args) {
        s = 'a';
        System.out.println(s);
    }
}
```

它将产生如下输出:

a

假定创建了类 Super 的新版本:

```
class Super { final static char s = 'b'; }
```

如果重新编译 Super, 但不重新编译 Test, 那么一起运行新的二进制代码与 Test 的现有二进制代码将会导致一个 `IllegalAccessError`。

删除关键字 final 或者把值更改为字段初始化的值, 不会中断与现有二进制代码的兼容性。

如果字段是一个常变量 (4.12.4 节), 那么删除关键字 final 或更改其值通过导致预先存在的二进制代码不会运行, 将不会中断与它们的兼容性, 但是它们将不会看到字段使用的任何新值, 除非重新编译它们。即使使用自身不是一个编译时常量表达式 (15.28 节), 也是如此。

如果编译并执行下列示例:

```
class Flags { final static boolean debug = true; }
class Test {
    public static void main(String[] args) {
        if (Flags.debug)
            System.out.println("debug is true");
    }
}
```

将产生以下输出:

debug is true

假定创建了类 Flags 的新版本:

```
class Flags { final static boolean debug = false; }
```

如果重新编译 Flags, 但不重新编译 Test, 那么一起运行新的二进制代码与 Test 现有的二进制代码将产生以下输出:

debug is true

因为 `debug` 的值是一个编译时常量，并且可以用在编译 `Test` 中，而不会产生一个指向类 `Flags` 的引用。

如第 14.21 节末尾所讨论的，这个结果是支持条件编译这一决定的副作用。

如果把 `Flags` 更改为一个接口，这种行为也不会发生改变，例如，在下面修改过的示例中：

```
interface Flags { boolean debug = true; }
class Test {
    public static void main(String[] args) {
        if (Flags.debug)
            System.out.println("debug is true");
    }
}
```

（需要内联常量的一种原因是：switch 语句在每个 case 上都需要常量，并且任何两个这样的常量值都不相同。编译器在编译时会检查重复的常量值；class 文件格式不会执行 case 值的符号链接。）

在广泛分发的代码中，避免具有“非恒定常量”这一问题的最佳方式是：仅当值确实永远都不太可能改变时，才将其声明为编译时常量。除了针对真正的数学常量之外，我们建议源代码应该极少使用声明为 `static` 和 `final` 的类变量。如果需要 `final` 的只读性质，更好的选择是声明一个 `private static` 变量，以及一个用于获取其值的合适方法。因此，我们建议使用：

```
private static int N;
public static int getN() { return N; }
```

而不是使用：

```
public static final int N = ...;
```

如果 `N` 不必是只读的，下面的语句没有任何问题：

```
public static int N = ...;
```

作为一种一般性的规则，我们还建议只在接口中声明真正的常量值。我们指出（但不建议）如果可以更改接口的基本类型的字段，那么习惯上可以把它的值表示如下：

```
interface Flags {
    boolean debug = new Boolean(true).booleanValue();
}
```

确保这个值不是一个常量。对于其他的基本类型，存在类似的编程风格。

要指出的另一件事情是：具有常量值的 `static final` 字段（无论是基本类型，还是 `String` 类型）看上去决不应该具有针对其类型的默认初始值（4.12.5 节）。这意味着所有这类字段在类初始化期间都会首先被初始化（8.3.2.1 节、9.3.1 节、12.4.2 节）。

### 13.4.10 static 字段

对于一个没有被声明为 `private` 的字段，如果以前没有将其声明为 `static`，并且现

在将其更改成声明为 `static` (反之亦然), 那么如果该字段是由预先存在的二进制代码使用的, 并且该二进制代码需要另一种类型的字段, 则会导致一个链接时错误 (确切地讲是一个 `IncompatibleClassChangeError`)。在广泛分发的代码中不建议执行这种更改。

### 13.4.11 `transient` 字段

添加或删除字段的 `transient` 修饰符不会中断与预先存在的二进制代码的兼容性。

### 13.4.12 方法和构造函数声明

如果由于某个调用以前引用了一个具有不兼容类型的超类的方法或构造函数, 而使得不再能够重新编译某个类型, 那么添加方法或构造函数声明到类中将不会中断与任何预先存在的二进制代码的兼容性。以前编译的具有这种引用的类将继续引用超类中声明的方法或构造函数。

假定一个引用指向一个具有合格类型  $T$  的方法  $m$ 。进一步假定  $m$  事实上是  $T$  的超类  $S$  中声明的一个实例 (或 `static`) 方法。如果将具有与  $m$  相同的签名和返回类型的类型  $X$  的新方法添加到  $S$  的子类中, 并且该子类是  $T$  的超类或  $T$  本身, 那么就可能发生链接错误。除了上述情况外, 仅当下列条件之一成立时, 才会发生这种链接错误:

- 与旧方法相比, 新方法具有较少的访问权限。
- 新方法是 `static` (或实例) 方法。

从类中删除一个方法或构造函数将中断与任何预先存在的引用该方法或构造函数的二进制代码的兼容性; 并且在链接这样一个来自预先存在的二进制代码的引用时, 将会抛出一个 `NoSuchMethodError`。仅当超类中没有声明具有匹配的签名和返回类型的方法时, 才会发生这种错误。

如果类的源代码中不包含声明的构造函数, Java 编译器就会自动提供一个无参构造函数。添加一个或多个构造函数声明到这样一个类的源代码中, 将会阻止自动提供这个默认构造函数, 从而有效地删除了一个构造函数, 除非新构造函数之一也没有参数, 从而可以代替默认构造函数。对于自动提供的无参构造函数, 将会给予其与其声明所在的类相同的访问修饰符, 因此, 如果将要保持与预先存在的二进制代码的兼容性, 那么任何替代的构造函数都应该具有同样多或更多的访问权限。

### 13.4.13 方法和构造函数的形式类型参数

重命名一个被声明为方法或构造函数的形式类型参数的类型变量 (4.4 节), 不会对预先存在的二进制代码产生任何影响。添加或删除类型参数本质上不会牵涉到二进制兼容性。



注意, 如果在方法或构造函数的类型中使用这种类型变量, 则通常可能牵涉到更改上述类型。

更改类型参数的第一次绑定，将会更改在它自己的类型中使用那个类型变量的任何成员的擦除（4.6 节），并且这可能会影响二进制兼容性。确切地讲：

- 如果把类型参数用作字段的类型，其效果就好像是删除了这个字段，并添加了一个同名字段，其类型是该类型变量新的擦除。
- 如果把类型变量用作方法的任何形参的类型，但是不用作返回类型，其效果就好像是删除了那个方法，并用一个除上述形参的类型以外其他方面完全相同的新方法代替它，这些形参现在把该类型变量新的擦除作为它们的类型。
- 如果把类型变量用作方法的返回类型，但是不用作方法的任何形参的类型，其效果就好像是删除了那个方法，并用一个除返回类型以外其他方面完全相同的新方法代替它，其返回类型现在是该类型变量新的擦除。
- 如果把类型变量用作方法的返回类型，以及方法的某些形参的类型，其效果就好像是删除了那个方法，并用一个新方法代替它，除返回类型和上述形参的类型之外，这个新方法与原来的方法完全一样，其返回类型现在是该类型变量新的擦除，上述形参现在则把该类型变量新的擦除作为它们的类型。

更改任何其他绑定不会对二进制兼容性产生任何影响。

#### 13.4.14 方法和构造函数的参数

更改方法或构造函数的形参名称不会影响预先存在的二进制代码。更改方法的名称、方法或构造函数的形参的类型，或者在方法或构造函数声明中添加或删除一个参数，将会创建具有新签名的方法或构造函数，并且具有删除带有旧签名的方法或构造函数以及添加带有新签名的方法或构造函数的组合效果（13.4.12 节）。

出于二进制兼容性考虑，添加或删除其签名涉及变量类型（4.4 节）或参数化类型（4.5 节）的方法或构造函数 *m* 等价于添加（或删除）一个在其他方面等价的方法，并且其签名是 *m* 的签名的擦除（4.6 节）。

#### 13.4.15 方法的结果类型

更改方法的结果类型，用 `void` 代替结果类型，或者用结果类型代替 `void`，具有删除旧方法以及添加一个具有新结果类型或最新 `void` 结果的新方法的组合效果（参见第 13.4.12 节）。

出于二进制兼容性考虑，添加或删除其返回类型涉及变量类型（4.4 节）或参数化类型（4.5 节）的方法或构造函数 *m* 等价于添加（或删除）一个在其他方面等价的方法，并且其返回类型是 *m* 的返回类型的擦除（4.6 节）。

#### 13.4.16 `abstract` 方法

将声明为 `abstract` 的方法更改成不再被声明为 `abstract` 不会中断与预先存在的二进制代码的兼容性。

将未声明为 `abstract` 的方法更改成声明为 `abstract` 将会中断与以前调用该方法预先存在的二进制代码的兼容性，并引发一个 `AbstractMethodError`。



如果编译并执行下面的示例程序：

```
class Super { void out() { System.out.println('Out'); } }
class Test extends Super {
    public static void main(String[] args) {
        Test t = new Test();
        System.out.println('Way ');
        t.out();
    }
}
```

则将输出如下结果：

```
Way
Out
```

假定创建了类 Super 的新版本：

```
abstract class Super {
    abstract void out();
}
```

如果重新编译 Super，但不重新编译 Test，那么一起运行新的二进制代码与 Test 现有的二进制代码将会导致一个 `AbstractMethodError`，因为类 Test 没有方法 out 的实现，因此是（或者应该是）抽象的。

### 13.4.17 final 方法

将非 final 的实例方法更改成 final 可能中断与现有二进制代码的兼容性，这取决于重写方法的能力。如果编译并执行下列测试程序：

```
class Super { void out() { System.out.println('out'); } }
class Test extends Super {
    public static void main(String[] args) {
        Test t = new Test();
        t.out();
    }
    void out() { super.out(); }
}
```

则将会输出如下结果：

```
out
```

假定创建了类 Super 的新版本：

```
class Super { final void out() { System.out.println('!'); } }
```

如果重新编译 Super，但不重新编译 Test，那么一起运行新的二进制代码与 Test 现有的二进制代码将会导致一个 `VerifyError`，因为类 Test 不正确地尝试重写实例方法 out。

将一个非 final 的类（static）方法更改为 final 不会中断与现有二进制代码的兼容性，因为该方法不能被重写。



从方法中删除 `final` 修饰符不会中断与预先存在的二进制代码的兼容性。

#### 13.4.18 native 方法

添加或删除方法的 `native` 修饰符不会中断与预先存在的二进制代码的兼容性。

对不会重新编译的预先存在的 `native` 方法的类型执行更改所产生的影响超出了本规范的范围，并且应该与实现的描述一起提供。鼓励（但不要求）实现以限制这种影响的方式来实现 `native` 方法。

#### 13.4.19 static 方法

如果以前把未声明为 `private` 的方法声明为 `static`（也就是说，声明为一个类方法），并且现在改为不将其声明为 `static`（也就是说，声明一个实例方法）（反之亦然），那么如果这些方法被预先存在的二进制代码使用，则可能会中断与预先存在的二进制代码的兼容性，从而导致一个链接时错误，即一个 `IncompatibleClassChangeError`。对于广泛分发的代码，不建议执行这种更改。

#### 13.4.20 synchronized 方法

添加或删除方法的 `synchronized` 修饰符不会中断与现有二进制代码的兼容性。

#### 13.4.21 方法或构造函数的 `throws` 子句

更改方法或构造函数的 `throws` 子句不会中断与现有二进制代码的兼容性；仅会在编译时检查这些子句。

#### 13.4.22 方法和构造函数的主体

更改方法或构造函数主体不会中断与预先存在的二进制代码的兼容性。

我们指出：编译器不能在编译时扩展一个内联方法。

方法上的关键字 `final` 并不意味着该方法可以被安全内联；这只意味着该方法不能被重写。仍有可能在链接时提供该方法的新版本。此外，出于反射的目的，必须保留原始程序的结构。

一般来讲，我们建议实现后期绑定（运行时）的代码生成和优化。

#### 13.4.23 方法和构造函数重载

添加重载现有方法或构造函数的新方法或构造函数，不会中断与预先存在的二进制代码的兼容性。用于每个调用的签名是在编译这些现有的二进制代码时确定的；因此，不会使用最近添加的方法或构造函数，即使它们的签名是合适的，并且与最初选择的签名相比更明确。

虽然添加新的重载方法或构造函数可能会在下一次编译类或接口时引发编译时错误[因为没有最明确的方法或构造函数(15.12.2.5节)],但是在执行程序时不会发生这种错误,因为在执行时不会进行重载解析。

如果编译并执行下列示例程序:

```
class Super {
    static void out(float f) { System.out.println("float"); }
}
class Test {
    public static void main(String[] args) {
        Super.out(2);
    }
}
```

则会产生如下输出:

```
float
```

假定创建了类 Super 的新版本:

```
class Super {
    static void out(float f) { System.out.println("float"); }
    static void out(int i) { System.out.println("int"); }
}
```

如果重新编译 Super, 但不重新编译 Test, 那么一起运行新的二进制代码与 Test 现有的二进制代码仍会产生如下输出:

```
float
```

但是, 如果随后使用这个新的 Super 重新编译 Test, 则会产生如下输出:

```
int
```

这可能与上一个示例中缺乏经验的人所预期的结果一样。

#### 13.4.24 方法重载

如果把一个实例方法添加到一个子类中, 并且它重写了某个超类中的一个方法, 那么将会在预先存在的二进制代码中通过方法调用找到该子类方法, 并且这些二进制代码不会受到影响。如果将一个类方法添加到某个类中, 那么将不会找到该方法, 除非引用的合格类型是子类类型。

#### 13.4.25 静态初始化语句

添加、删除或更改类的静态初始化语句(8.7节)不会影响预先存在的二进制代码。

#### 13.4.26 枚举的演变

在枚举类型中添加或重排常量的顺序将不会中断与预先存在的二进制代码的兼容性。

如果预编译的二进制代码试图访问一个不再存在的枚举常量，那么客户将会在运行时失败，并且会抛出一个 `NoSuchFieldError`。因此，对于广泛分布的枚举，不建议执行这种更改。

在所有其他方面，针对枚举的二进制兼容性规则与那些针对类的二进制兼容性规则完全一样。

## 13.5 接口的演变

本节描述了在预先存在的二进制代码上更改接口的声明及其成员所产生的影响。

### 13.5.1 public 接口

将未被声明为 `public` 的接口更改成声明了的 `public`，不会中断与预先存在的二进制代码的兼容性。

如果把声明为 `public` 的接口更改成未声明 `public`，那么如果在链接预先存在的二进制代码时需要访问该接口类型，但是不再具有访问它的权限，则会抛出一个 `IllegalAccessError`，因此，对于广泛分发的接口，不建议使用这种更改。

### 13.5.2 超接口

更改接口层次结构将会以与更改类层次结构相同的方式引发错误，如第 13.4.4 节所述。特别地，对于导致类的任何以前的超接口不再是一个超接口的更改，可能会中断与预先存在的二进制代码的兼容性，从而导致一个 `VerifyError`。

### 13.5.3 接口成员

添加一个方法到接口中不会中断与预先存在的二进制代码的兼容性。添加到 `C` 的超接口中的字段可能会隐藏一个继承自 `C` 的超类的字段。如果原始引用指向一个实例字段，则会引发一个 `IncompatibleClassChangeError`。如果原始引用是一个赋值，则会引发一个 `IllegalAccessError`。

从接口中删除一个成员可能会引发预先存在的二进制代码中的链接错误。

如果编译并执行下列示例程序：

```
interface I { void hello(); }
class Test implements I {
    public static void main(String[] args) {
        I anI = new Test();
        anI.hello();
    }
    public void hello() { System.out.println('hello'); }
}
```

则会产生如下输出：

```
hello
```

假定编译了接口 I 的新版本：

```
interface I { }
```

如果重新编译 I，但不重新编译 Test，那么一起运行新的二进制代码与 Test 现有的二进制代码将会导致一个 `NoSuchMethodError`（在某些早期的实现中，仍会执行这个程序；不会正确地检测到方法 `hello` 不再存在于接口 I 中的事实）。

### 13.5.4 接口的形式类型参数

更改接口的形式类型参数的效果与对类的形式类型参数执行类似更改的效果相同。

### 13.5.5 字段声明

更改接口中的字段声明的考虑事项与更改类中的 `static final` 字段的那些考虑事项相同，如第 13.4.8 节和 13.4.9 节所述。

### 13.5.6 抽象方法声明

更改接口中的抽象方法声明的考虑事项与更改类中的 `abstract` 方法的那些考虑事项相同，如第 13.4.14 节、13.4.15 节、13.4.21 节和 13.4.23 节所述。

### 13.5.7 注释类型的演变

注释类型的行为方式与其他任何接口完全一样。在注释类型中添加或删除一个元素类似于添加或删除一个方法。有许多重要的考虑事项支配着对注释类型所做的其他更改，但是这些更改不会通过 Java 虚拟机对二进制代码的链接产生任何影响。相反，这些更改会影响操纵注释的反射 API 的行为。有关这些 API 的文档详细说明了当对底层注释类型执行各种更改时它们的行为。

添加或删除注释不会对以 Java 编程语言编写的程序的二进制表示的正确链接产生任何影响。

瞧！请看，杰出的科学！……  
然而，瞧！精神——在所有科学之上……  
为了它，部分转变为永久流动  
为了它，现实转变为理想趋势  
为了它，神秘的进化……  
——沃尔特·惠特曼《自己之歌》（1874 年）

## 块 和 语 句

他不仅仅是旧部件的芯片，而且就是旧部件本身。  
——Edmund Burke, 《On Pitt's First Speech》

程序的执行顺序是由语句控制的，这些语句是针对其效果而执行的，并且没有值。

一些语句将其他语句作为其结构的一部分进行包含；这样的一些其他语句是语句的子语句。我们说如果没有不同于  $S$  和  $U$  的语句使得  $S$  包含  $T$ ，且  $T$  包含  $U$ ，那么语句  $S$  就直接包含语句  $U$ 。以相同的方式，这些语句可以将表达式（第 15 章）作为其结构的一部分进行包含。

本章的第一节讨论了语句（14.1 节）的正常和突然结束的不同。其余部分大多解释了各种语句，详细描述了它们的正常行为和异常结束的特殊处理。

首先对块进行解释（14.2 节），接着解释本地类声明（14.3 节）和局部变量声明语句（14.1 节）。

接下来，解释回避熟悉的“dangling else”问题的方法。

本章的最后一节（14.21 节）解决了每条语句在某个技术意义上都是可到达的需求。

## 14.1 语句的正常结束和突然结束

“波洛的突然离开完全激起了我们的兴趣。”  
——阿加莎·克里斯蒂《斯泰尔斯庄园奇案》（1920），第 12 章

每个语句都有一个正常的执行模式，在这样的模式下，会执行某些计算步骤。接下来的几段描述了每种语句的正常执行模式。如果所有的步骤都正常执行，没有突然结束的迹象，则称这个语句正常结束。但某些事件可能阻止语句正常结束。

- `break`（14.15 节）、`continue`（14.16 节）和 `return`（14.17 节）语句导致传输了可以防止包含它们的语句正常结束的控制。
- 某个表达式的求值可能从 Java 虚拟机抛出异常；这些表达式在第 15.6 节中进行了总结。显示的 `throw`（14.18 节）语句也导致了一个异常。异常导致传输了可以防止语句正常结束的控制。

如果这样的事件发生，那么一条或多条语句的执行可能在其正常执行模式的所有步骤

完成前终止。这样的语句就是突然结束。

突然结束总有一个相关的原因，为下列原因之一：

- 没有标签的 `break`
- 有给定标签的 `break`
- 没有标签的 `continue`
- 有给定标签的 `continue`
- 没有值的 `return`
- 有给定值的 `return`
- 具有指定值的 `throw`，包括 Java 虚拟机抛出的异常

术语“正常结束”和“突然结束”也应用于表达式的求值（15.6 节）。一个表达式可能突然结束的惟原因是抛出了一个异常，要么由于一个有给定值的 `throw`（14.18 节），要么由于一个运行时异常或错误（11 章、15.6 节）

如果一条语句计算一个表达式，表达式的突然结束始终导致语句的立即突然结束，而且原因是相同的。执行的正常模式中的所有后续步骤都没有被执行。

除非本章中另有说明，否则子语句的突然结束导致语句本身的突然结束，其原因是一样的，并且语句执行的正常模式中所有后续步骤将不被执行。

除非另有说明，否则如果语句计算的所有表达式及它执行的所有子表达式正常结束，那么语句就正常结束。

## 14.2 块

他的心就像他帽子的式样一般，时时刻刻会起变化的。  
——威廉·莎士比亚，《无事生非》（1623）第一幕第一场

块是放在大括号中的一连串语句、本地类声明和局部变量声明语句。

*Block:*

*{ BlockStatements<sub>opt</sub> }*

*BlockStatements:*

*BlockStatement*

*BlockStatements BlockStatement*

*BlockStatement:*

*LocalVariableDeclarationStatement*

*ClassDeclaration*

*Statement*

块是通过按从第一个到最后一个的顺序（左到右）执行每个局部变量声明语句和其他语句来执行的。如果所有的块语句正常结束，那么块就正常结束。如果出于任何原因，这些块语句的任何语句突然结束，那么块就出于相同的原因突然结束。



## 14.3 本地类声明

本地类是一个嵌套类（第 8 章），该嵌套类不是任何类的一个成员，并且具有一个名称。所有的本地类都是内部类（8.1.3 节）。每个本地类声明语句直接由一个块包含。本地类声明语句可以自由地与块的其他种语句混合。

直接由块（14.2 节）封闭的本地类的作用域是直接封闭的块的其余部分，包括它自己的声明。通过在 switch 块语句组（14.11 节）中直接封闭的本地类的作用域是直接封闭 switch 块语句组的其余部分，包括它自己的类声明。

本地类 C 的名称可能不可以重新声明为 C 的作用域中的直接封闭方法、构造函数或初始化程序块的一个本地类，否则就会发生编译时错误。但本地类声明可以嵌套在本地类声明作用域的任何地方进行屏蔽（6.3.1 节）。本地类不能有一个规范的名称，它也没有一个完全限定名称。

如果本地类声明包含下面访问修饰符的任何之一，那么它就是一个编译时错误：  
`public`、`protected`、`private` 或 `static`。

下面是例示上面给出的规则的几个方面的例子：

```
class Global {
    class Cyclic {}
    void foo() {
        new Cyclic(); // create a Global.Cyclic
        class Cyclic extends Cyclic(); // circular definition
    {
        class Local{};
        {
            class Local{}; // compile-time error
        }
        class Local{}; // compile-time error
        class AnotherLocal {
            void bar() {
                class Local {}; // ok
            }
        }
    }
    class Local{}; // ok, not in scope of prior Local
}
```

方法 `foo` 的第一个语句创建了成员类 `Global.Cyclic` 的一个实例，而不是创建本地类 `Cyclic` 的一个实例，因为本地类声明仍不在此作用域中。

本地类作用域包含它自己的声明（而不是它的体）的事实意味着本地类 `Cyclic` 的定义确实是循环的，因为它扩展了自身，而不是扩展了 `Global.Cyclic`。因此，本地类 `Cyclic` 的声明将在编译时被拒绝。

由于本地类名称不能在相同的方法（或构造函数或初始化，等其他可能的情形）重新进行声明，所以 `Local` 的第二和第三个声明导致了编译时错误。但 `Local` 可以在另一个

(更加深层嵌套的)类(比如 AnotherLocal)的上下文中进行重新声明。

Local 的第 4 个和最后一个声明是合法的,因此它在 Local 任何在前声明的作用域之外发生。

## 14.4 局部变量声明语句

局部变量声明语句声明了一个或多个局部变量名称。

*LocalVariableDeclarationStatement:*

*LocalVariableDeclaration ;*

*LocalVariableDeclaration:*

*VariableModifiers Type VariableDeclarators*

下面对第 8.3 节的重复,使得这里的陈述更加清楚:

*VariableDeclarators:*

*VariableDeclarator*

*VariableDeclarators , VariableDeclarator*

*VariableDeclarator:*

*VariableDeclaratorId*

*VariableDeclaratorId = VariableInitializer*

*VariableDeclaratorId:*

*Identifier*

*VariableDeclaratorId [ ]*

*VariableInitializer:*

*Expression*

*ArrayInitializer*

每个局部变量声明语句直接由一个块包含。局部变量声明语句可以自由地与块中的其他种语句相互混合。

局部变量声明也可以出现在 for 语句的头中(14.14 节)。在这种情形中,它被执行的方式就像它是局部变量声明语句那样。

### 14.4.1 局部变量声明符和类型

局部变量声明中的每个声明符声明了一个局部变量,其名称是出现描述符中的标识符。

如果可选的关键字 final 出现在声明符的开始,则被声明的变量是最终变量(4.12.4 节)。

如果局部变量声明上的注释 *a* 对应于注释类型 *T*,并且 *T* 有一个对应于 annotation.Target 的(元)注释 *m*,那么 *m* 必须有一个其值为 annotation.ElementType.LOCAL\_

VARIABLE 的元素，否则会发生编译时错误。第 9.7 节进一步描述了注释修饰符。

变量的类型是由出现在局部变量声明中的 *Type*，再跟上声明符中 *Identifier* 后面的任何大括号对进行表示的。

因此，局部变量声明：

```
int a, b[], c[][];
```

等价于下面的声明序列：

```
int a;
int[] b;
int[][] c;
```

在声明符中，中括号是允许的，以便于与 C 和 C++ 的传统兼容。但一般规则也意味局部变量声明：

```
float[][] f[], g[][][], h[]; // Yechh1
```

等价于下面的声明序列：

```
float[][][] f;
float[][][] g;
float[][] h;
```

我们不推荐这种数组声明的“混合符号”。

类型为 `float` 的局部变量始终包含一个值，这个值是浮点值集合（4.2.3 节）的一个元素；同样，类型为 `double` 的局部变量始终包含一个值，这个值是双精度值集合的一个元素。不允许让类型为 `float` 的局部变量包含浮点扩展指数值集合的一个元素，该元素也不是浮点数值集合的一个元素，也不允许类型为 `double` 的局部变量包含双精度扩展指数值集合的一个元素，该元素也不是双精度值集合的一个元素。

## 14.4.2 局部变量声明的作用域

块（14.4.2 节）中的局部变量声明的作用域是声明出现所在的块的其余部分，从它自己初始化程序（14.4 节）开始，并包括局部变量声明语句右边的任何进一步声明符。

局部变量 *v* 的名称不可以被重新声明为 *v* 的作用域中的直接封闭方法、构造函数或初始化程序块的局部变量，否则会发生编译时错误。局部变量 *v* 的名称不可以被重新声明为 *v* 的作用域中的直接封闭方法、构造函数或初始化块中的 `try` 语句中的 `catch` 子句的异常参数，否则会发生编译时错误。但一个方法或初始化程序块的变量可以嵌套在局部变量作用域中的类声明内部的任何地方而成为屏蔽（6.3.1 节）。

局部变量不可以使用限定名称，只可以使用一个简单的名称进行引用。

下面的例子：

```
class Test {
    static int x;
    public static void main(String[] args) {
        int x = x;
    }
}
```

导致一个编译时错误，因为 `x` 的初始化是在将 `x` 声明为局部变量的作用域的，并且局部 `x` 仍然没有一个值，并且不可以使用。

下面的程序进行了编译：

```
class Test {
    static int x;
    public static void main(String[] args) {
        int x = (x=2)*2;
        System.out.println(x);
    }
}
```

因为局部变量 `x` 在它被使用之前是明确赋值的。输出如下：

4

下面是另一个例子：

```
class Test {
    public static void main(String[] args) {
        System.out.print("2+1=");
        int two = 2, three = two + 1;
        System.out.println(three);
    }
}
```

上面的程序正确编译并产生输出：

2+1=3

`three` 的初始化程序可以正确地引用在前面的声明符中声明的变量 `two`，并且下一行的方法调用可以正确引用块中前面声明的变量 `three`。

在 `for` 语句中声明的一个局部变量的作用域是 `for` 语句的其余部分，包括它自己的初始化程序。

如果将一个标识符声明为相同的方法、构造函数或初始化程序块的一个局部变量出现在相同名称的参数或局部变量中，就会发生编译时错误。

下面的例子没有进行编译：

```
class Test {
    public static void main(String[] args) {
        int i;
        for (int i = 0; i < 10; i++)
            System.out.println(i);
    }
}
```

此限制有助于检测一些特别隐蔽的 `bug`。通过局部变量在成员的屏蔽上进行类似的限制是不切实际的决策，因为在超类中添加一个成员可能导致子类重命名局部变量。相关的考虑会根据嵌套类的成员在局部变量的屏蔽上进行限制，或者通过在没有吸引力的嵌套类中声明的局部变量在局部变量的屏蔽上进行限制。因此，下面的示例无错地进行了编译：

```
class Test {
    public static void main(String[] args) {
        int i;
        class Local {
            {
                for (int i = 0; i < 10; i++)
                    System.out.println(i);
            }
        }
        new Local();
    }
}
```

另一方面，具有相同名称的局部变量可以在两个不同的块或 for 语句（其中任何一个都不包含另外一个）中进行声明。从而：

```
class Test {
    public static void main(String[] args) {
        for (int i = 0; i < 10; i++)
            System.out.print(i + " ");
        for (int i = 10; i > 0; i--)
            System.out.print(i + " ");
        System.out.println();
    }
}
```

编译并没有错误，执行时产生下面的输出：

```
0 1 2 3 4 5 6 7 8 9 10 9 8 7 6 5 4 3 2 1
```

### 14.4.3 通过局部变量的名称屏蔽

如果将一个名称声明为一个已经声明为字段名称的局部变量，那么在整个局部变量的作用域，那个外部的声明就被屏蔽（6.3.1 节）。同样，如果一个名称已经被声明为一个变量或参数名称，那么在局部变量的整个作用域外部声明就被屏蔽（前提是在第 14.4.2 节的规则下，屏蔽没有导致编译时错误）。屏蔽的名称有时可以使用恰当限定的名称进行访问。

例如，关键字 `this` 可用于访问一个屏蔽字段 `x`，方法是使用 `this.x` 格式。的确，该语法通常出现在构造函数中（8.8 节）：

```
class Pair {
    Object first, second;
    public Pair(Object first, Object second) {
        this.first = first;
        this.second = second;
    }
}
```

在本例中，构造函数接收具有与初始化的字段相同名称的参数。相对于需要为参数设计名称，这是更加简单的，并且在样式化的上下文中，这不会太混淆。但通常，这被认为是不好的样式，这样会有与字段相同的名称。

#### 14.4.4 局部变量声明的执行

局部变量声明语句是一个可执行语句。每当它被执行时，声明符就按照从左到右的顺序进行处理。如果声明符有一个初始化表达式，那就计算表达式，并且它的值被赋予变量。如果声明符没有初始化表达式，那么 Java 编译器必须使用第 16 章中给出的算法证明：对变量的每个引用之前必须执行对变量的赋值。如果不是这样的一种情形，那么就发生编译时错误。

只有前面的初始化表达式正常结束时每个初始化（除了第一个）才会被执行。只有最后一个初始化表达式正常结束时，局部变量声明的执行才正常结束；如果局部变量声明没有包含初始化表达式，那么执行它始终正常结束。

#### 14.5 语句

在 Java 编程语言中，有许多种语句。其中大多数对应于 C 和 C++ 语言中的语句，但有一些则是惟一的。

如在 C 和 C++ 中一样，Java 编程语言的 if 语句遭受了所谓的“虚悬 else 问题”，如下面的这个令人误解的例子所示：

```
if (door.isOpen())
    if (resident.isVisible())
        resident.greet("Hello!");
else door.bell.ring(); //A "dangling else"
```

这里的问题是外部的 if 语句和内部的 if 语句可能确实拥有 else 语句。在本例中，可以猜出该语法有意让 else 语句属于外部 if 语句。Java 编程语言（像 C 和 C++ 及它们之前的许多编程语言一样）任意颁布 else 子句属于它可能属于的最内部 if。此规则是通过下面的语法捕获的：

*Statement:*

- StatementWithoutTrailingSubstatement*
- LabeledStatement*
- IfThenStatement*
- IfThenElseStatement*
- WhileStatement*
- ForStatement*

*StatementWithoutTrailingSubstatement:*

- Block*
- EmptyStatement*
- ExpressionStatement*
- AssertStatement*



*SwitchStatement*  
*DoStatement*  
*BreakStatement*  
*ContinueStatement*  
*ReturnStatement*  
*SynchronizedStatement*  
*ThrowStatement*  
*TryStatement*

*StatementNoShortIf:*

*StatementWithoutTrailingSubstatement*  
*LabeledStatementNoShortIf*  
*IfThenElseStatementNoShortIf*  
*WhileStatementNoShortIf*  
*ForStatementNoShortIf*

下面重复了第 14.9 节，以便使这里的陈述更加清晰：

*IfThenStatement:*

*if ( Expression ) Statement*

*IfThenElseStatement:*

*if ( Expression ) StatementNoShortIf else Statement*

*IfThenElseStatementNoShortIf:*

*if ( Expression ) StatementNoShortIf else StatementNoShortIf*

语句因而在语法上被分成两类：可能在没有 *else* 语句的 *if* 语句（“一条短的 *if* 语句”）中结束的语句，以及确实没有在短的 *if* 语句中结束的语句。只有确实没有在短的 *if* 语句中结束的语句才可以在确实没有 *else* 子句的 *if* 语句中的关键字 *else* 之前作为直接子句出现。

简单的规则防止了“虚悬 *else*”问题。具有“没有短的 *if*”限制的语句的执行行为等同于相同种语句的执行行为，但没有“没有短的 *if*”限制；这些区别要彻底地进行区分，以解决句法困难。

## 14.6 空语句

谁看到过这样一颗空洞的心，吼起来却这样有劲？

不过俗话说得好：“喊得越响，肚里越空。”

——威廉·莎士比亚，《亨利五世》（1623）第四幕第四场

一条空语句将什么都不做。

*EmptyStatement:*

;

空语句的执行始终正常完成。

## 14.7 标签语句

在 5 分钟内我就爬到了马背上，对自己的全套装备相当满意。

我没时间做个标签写上“这是一匹马”，

所以，假如别人认为这是一头羊，我也没办法。

——马克·吐温，《苦行记》(1871)

语句可以有 *label* 前缀。

*LabeledStatement:*

*Identifier* : *Statement*

*LabeledStatementNoShortIf:*

*Identifier* : *StatementNoShortIf*

*Identifier* 被声明成直接包含的 *Statement* 的标签。

C 和 C++ 不一样，Java 编程语言没有 *goto* 语句；标识符语句是和 *break* (4.5 节) 或 *continue* (14.16 节) 语句一起使用的，这些语句出现在标签语句中的任何地方。

令 *l* 为一个标签，并令 *m* 为一个直接封闭的方法、构造函数、实例初始化程序或静态初始化程序。如果 *l* 屏蔽了直接封闭在 *m* 中的另一个标签的声明，那么它就是一个运行时错误。

没有针对使用相同的标识符作为标签和作为包、类、接口、方法、字段、参数或局部变量的名称的限制。使用标识符来标签语句不会混淆 (6.3.2 节) 具有相同名称的包、类、接口、方法、字段、参数或局部变量。将标识符用作类、接口、方法、字段、局部变量或用作异常处理程序 (14.20 节) 的参数，不会混淆具有相同名称的语句标签。

标签语句是通过执行直接包含的 *Statement* 执行的。如果语句是通过 *Identifier* 加标签的，并且包含的 *Statement* 由于具有相同 *Identifier* 的 *break* 突然结束，那么标签语句就突然结束。在 *Statement* 的所有其他突然结束的情形中，标签语句出于相同的原因突然结束。

## 14.8 表达式语句

某些类别的表达式通过在它们后面加个分号可以用作语句：

*ExpressionStatement:*

*StatementExpression* ;

*StatementExpression:*

*Assignment*

*PreIncrementExpression*  
*PreDecrementExpression*  
*PostIncrementExpression*  
*PostDecrementExpression*  
*MethodInvocation*  
*ClassInstanceCreationExpression*

表达式语句是通过求表达式的值执行的：如果表达式具有一个值，那么该值就被丢弃。当且仅当表达式的求值正常结束时，表达式语句的执行才正常结束。

与 C 和 C++ 不一样，Java 编程语言仅允许某些形式的表达式用作表达式语句。注意，Java 编程语言不允许“强制类型转换到 void”——void 不是一种类型——因此，如下所示的编写表达式语句的传统 C 语言技巧是行不通的：

```
(void) ... ; // incorrect!
```

另一方面，语言允许表达式语句中所有最有用的表达式，并且它不需要将方法调用用作表达式语句来调用 void 方法，因此几乎从不需要诸如此类的技巧。如果确实需要技巧，可以使用赋值语句（15.26 节）或局部变量声明语句（14.4 节）进行替代。

## 14.9 if 语句

if 语句允许语句的条件执行或两个语句的条件选择，执行其中一个或另一个，但不是两个都执行。

*IfThenStatement:*

*if ( Expression ) Statement*

*IfThenElseStatement:*

*if ( Expression ) StatementNoShortIf else Statement*

*IfThenElseStatementNoShortIf:*

*if ( Expression ) StatementNoShort else StatementNoShortIf*

*Expression* 必须具有类型 `boolean` 或 `Boolean`，否则会发出编译时错误。

### 14.9.1 if-then 语句

我得到了及早测试那句话的机会……  
——阿加莎·克里斯蒂，《斯泰尔斯庄园奇案》（1920）第 12 章

if-then 语句是通过首先计算 *Expression* 执行的。如果结果的类型是 `Boolean`，它就遵从于拆箱转换（5.1.8 节）。如果 *Expression* 的求值或后续拆箱转换（如果有的话）出于某个原因突然结束的话，那么 if-then 语句就出于相同的原因突然结束。否则，执行

通过基于结果值进行选择来继续：

- 如果值是 `true`，那么包含的 *Statement* 被执行；当且仅当 *Statement* 的执行正常结束时，`if-then` 语句才正常结束。
- 如果值是 `false`，那么就不采取任何进一步操作，并且 `if-then` 语句正常结束。

### 14.9.2 if-then-else 语句

你是否曾被迫作最终决定——  
对某人说是不再去管另外一个人？

——John Sebastian, 《Did You Ever Have to Make Up Your Mind?》

`if-then-else` 语句是通过首先计算 *Expression* 来执行的。如果结果的类型是 `Boolean`，那么它就遵从拆箱转换（5.1.8 节）。如果 *Expression* 的计算或后续拆箱转换（如果有的话）出于某个原因突然结束，那么 `if-then-else` 语句就出于相同的原因突然结束。否则，执行通过基于结果值做出选择以便继续：

- 如果值是 `true`，那么就执行第一个包含的 *Statement*（在 `else` 关键字前的那个语句）；当且仅当语句的执行正常结束时，`if-then-else` 语句正常结束。
- 如果值是 `false`，那么就执行第二个包含的 *Statement*（在 `else` 关键字后面的那个语句）；当且公当那条语句的执行正常结束时，`if-then-else` 语句正常结束。

## 14.10 assert 语句

断言是一个包含布尔表达式的语句。一个断言要么被启用，要么被禁用。如果断言被启用，那么断言的计算会导致布尔表达式的计算，并且在表达式求值为假的情况下会报告一个错误。如果断言被禁用了，那么断言的求值无论在什么情况下都不会起作用：

*AssertStatement*:

```
assert Expression1 ;
```

```
assert Expression1 : Expression2 ;
```

如果 *Expression1* 不具有类型 `boolean` 或 `Boolean`，那么它就是一个编译时错误。在 `assert` 语句的第二形式中，如果 *Expression2* 是 `void`（15.1 节），那么它就是一个编译时错误。

断言可以以每个类为基础进行启用或禁用。在类被初始化时（12.4.2 节），在类变量（8.3.2.1 节）的任何初字段始化器和静态初始化程序（8.7 节）的执行前，类的类加载器确定断言是否启用或禁用，如下面所描述。一旦一个类已经被初始化，那么它的断言状态（启用或禁用）没有更改。

### 讨论

有一种需要特殊处理的情形。回想一个类的断言状态是在它被初始化时进行设置的。

在初始化前可能要执行方法或构造函数（尽管通常不是所需的）。当类的层次结构在它的静态初始化中包含一个环状时，这种情况就可能发生了，比如下面的例子：

```
public class Foo {
    public static void main(String[] args) {
        Baz.testAsserts();
        // Will execute after Baz is initialized.
    }
}
class Bar {
    static {
        Baz.testAsserts();
        // Will execute before Baz is initialized!
    }
}
class Baz extends Bar {
    static void testAsserts(){
        boolean enabled = false;
        assert enabled = true;
        System.out.println("Asserts " +
                           (enabled ? "enabled" : "disabled"));
    }
}
```

调用 `Baz.testAsserts()` 导致 `Baz` 被初始化。在这种情形可能发生之前，`Bar` 必须被初始化。`Bar` 的静态初始化程序再次调用 `Baz.testAsserts()`。因为 `Baz` 的初始化已经通过当前线程在进行中，第二个调用立即执行，尽管 `Baz` 没有被初始化（JLS 12.4.2）。

如果 `assert` 语句在它的类被初始化之前执行了，如上面的例子所示，那么执行必须表现得像类中启用的断言那样。

## 讨论

换句话说，如果上面的程序在没有启用断言的情况下执行了，那么它必须输出：

```
Asserts enabled
Asserts disabled
```

当且仅当词汇上包含 `assert` 语句的最顶层类（第 8 章）启用断言时，该断言才是被启用的。顶层类是否启用断言，是在类被初始化（12.4.2 节）前通过它的定义类加载器确定的，并且在以后不能进行更改。

如果 `assert` 语句还没有被初始化的话，它导致封闭的顶层类（如果它存在的话）被初始化（12.4.1 节）。

### 讨论

注意，通过最顶层接口封闭的断言不会导致初始化。

通常，包括断言的最顶层类将已经被初始化。但是如果断言只位于静态嵌套类中，那么它可能还没有被初始化。

禁用的 `assert` 语句不起作用。尤其是 `Expression1` 和 `Expression2`（如果它存在的话）都没有被计算。禁用的 `assert` 语句的执行始终正常结束。

启用的 `assert` 语句是通过首先求 `Expression1` 的值进行计算的。如果结果的类型是 `Boolean`，那么它就遵从拆箱转换（5.1.8 节）。如果 `Expression1` 的求值或后续拆箱转换（如果有的话）出于某个原因突然结束的话，那么 `assert` 语句出于相同的原因突然结束。否则，执行通过基于 `Expression1` 的值做出选择以便继续：

- 如果值是 `true`，就不采取任何进一步操作，并且断言正常结束。
- 如果值为 `false`，那么执行行为取决于 `Expression2` 是否存在：
  - ◆ 若 `Expression2` 存在，那就计算该表达式。
    - ◇ 如果表达式出于某些原因突然结束，那么 `assert` 语句就出于相同的原因突然结束。
    - ◇ 如果计算正常结束，那么结果值使用字符串转换（15.18.1.1 节）转换到一个 `String`。
      - 若字符串转换出于某种原因而突然结束，那么 `assert` 语句就出于相同的原因突然结束。
      - 若字符串转换正常结束，那么就创建其“详细信息”是字符串转换的结果的 `AssertionError` 实例。
        - ◆ 如果实际创建出于某个原因突然结束，那么 `assert` 语句就出于相同的原因突然结束。
        - ◆ 如果实例创建正常结束，那么 `assert` 语句就通过抛出新创建的 `AssertionError` 对象突然结束。
  - ◆ 若 `Expression2` 不存在，那就创建没有“详细信息”的 `AssertionError` 实例。
    - ◇ 如果实例创建出于某个原因突然结束，那么 `assert` 语句就出于相同的原因突然结束。
    - ◇ 如果实例创建突然结束，那么 `assert` 语句就通过抛出新创建的 `AssertionError` 对象突然结束。

例如，在未从数据缓冲封送处理所有参数前，程序员可以断言保存在缓冲中的数据字节数是 0。通过验证布尔表达式确实为真，那么就确认了程序员对程序的了解，并增加了我们对程序没有 bug 的信心。



通常，断言检查是在程序开发和测试期间启用的，并针对部署进行禁用，以提高性能。

因为断言可以被禁用，所以程序必须假设在断言中包含的表达式将被计算。从而，这些布尔表达式一般应该不会有副作用：

对这样的—个布尔表达式求值应该不会影响在计算完成后可见的任何状态。让包含在断言中的—个布尔表达式有副作用并非是不合法的，但通常是不合适的，因为它导致程序行为的变化取决于断言是否是禁用或启用。

接下来的类似行，断言应该不会用于公共方法的参数检查。参数检查通常是方法的合同的一部分，并且该合同必须明确断言是启用的还是禁用的。

另一个与断言用于参数检查有关的问题是错误的参数会导致不恰当的运行时异常（比如 `IllegalArgumentException`、`IndexOutOfBoundsException` 或 `NullPointerException`）。断言失败将不会抛出一个合适的异常。另外，将断言用于公共方法上的参数检查不是非法的，但通常是不合适的。有意让 `AssertionError` 从未被捕获，但这样做是可能的，从而 `try` 语句的规则应该像 `throw` 语句的当前处理那样处理出现在 `try` 块中的断言。

## 14.11 switch 语句

给我找六七根山楂木的棍子来，要粗的，这些都太细了……  
——威廉·莎士比亚，《亨利八世》（1623）第五幕第四场

`switch` 语句根据表达式的值将控制传送到几个语句之一。

*SwitchStatement:*

`switch ( Expression ) SwitchBlock`

*SwitchBlock:*

`{ SwitchBlockStatementGroupsopt SwitchLabelsopt }`

*SwitchBlockStatementGroups:*

`SwitchBlockStatementGroup`

`SwitchBlockStatementGroups SwitchBlockStatementGroup`

*SwitchBlockStatementGroup:*

`SwitchLabels BlockStatements`

*SwitchLabels:*

`SwitchLabel`

`SwitchLabels SwitchLabel`

*SwitchLabel:*

`case ConstantExpression :`

`case EnumConstantName :`

default :

*EnumConstantName:*

*Identifier*

*Expression* 的类型必须是 char、byte、short、int、Character、Byte、Short、Integer 或一个枚举类型 (8.9 节), 否则会发生一个编译时错误。

switch 的体被称为开关块。开关块直接包含的语句可以用一个或多个 case 或者 default 标签进行标记。这些标签是与 switch 语句相关的, 它们是 case 标签中常量表达式 (15.28 节) 的值。

下面所列均必须为真, 否则将导致一个编译时错误:

- 与 switch 语句有关的每个常量表达式必须是可赋值 (5.2 节) 到 switch 表达式的类型。
- switch 标签均不是 null。
- 不会有两个与 switch 语句相关的 case 常量表达式具有相同的值。
- 至多一个 default 标签可以与同一个 switch 语句相关。

#### 讨论

禁止使用 null 作为开关标签防止了有人编写可能从未执行的代码。如果开关表达式是一个引用类型, 比如一个装箱基本类型或一个枚举, 那么如果表达式在运行时求值为 null, 就会发生运行时错误。

可以推断, 如果开关表达式的类型是一个枚举类型, 那么开关标签的可能值必须都是此种类型的枚举常量。

如果枚举值表达式上的一个开关缺少一个默认的情形, 或缺少一个或多个枚举类型的常量的情形, 就推荐 (但不是必需) 编译器提供一个警告 (如果表达式值求值为缺少的常量之一的話, 这样的一条语句将安静地不做任何事情)。

在 C 和 C++ 中, switch 语句的体可以是一条或多条具有 case 标签的语句, 这些标签不必直接由那条语句包含。考虑简单的循环:

```
for (I = 0; i < n; ++i) foo();
```

其中 n 是已知的正数。一个已知的 Duff 的设备的技术可以用在 C 或 C++ 中, 以便打开循环, 但在 Java 编程语言中, 这不是正确的代码。

```
int q = (n+7)/8;
switch (n%8) {
case 0: do {foo(); // Great C hack, Tom,
case 7: foo();      // but it's not valid here.
case 6: foo();
case 5: foo();
case 4: foo();
case 3: foo();
```

```
case 2: foo();  
case 1: foo();  
    } while (--q > 0);  
}
```

幸运的是，此技巧似乎并没有被广泛认知和使用。而且，现今更不需要它了：这种代码转换非常适合于最新式优化编译器。

当执行 `switch` 语句时，首先计算 *Expression*。如果 *Expression* 求值为 `null`，那么就抛出 `NullPointerException`，并且整个 `switch` 语句出于那种原因突然结束。否则，如果结果是引用类型，那么它就遵从拆箱转换（5.1.8 节）。如果 *Expression* 的求值或后续的拆箱转换（如果有的话）出于某个原因突然结束，那么 `switch` 语句就出于相同的原因突然结束。否则，执行通过下述方式继续：将 *Expression* 的值与每个 `case` 常量进行比较。然后进行下面的选择：

- 如果 `case` 常量之一等于表达式的值，那么我们就说 `case` 匹配，并且开关块中匹配 `case` 标签后面的所有语句（如果有的话）按顺序执行。如果这些语句正常结束，或者如果在匹配 `case` 标签后没有语句，那么整个 `switch` 语句正常结束。
- 如果没有 `case` 匹配，但有一个 `default` 标签，那么开关块中匹配 `default` 标签之后的所有语句（如果有的话）按顺序执行。如果所有的语句正常结束，或者如何 `default` 标签之后没有语句，那么整个 `switch` 语句正常结束。
- 如果没有 `case` 匹配，并且没有 `default` 标签，那么就不会采取进一步操作，并且 `switch` 语句正常结束。

如果 `switch` 语句的 *Block* 体直接包含的任何语句突然结束，那么它就如下所述进行处理：

- 如果 *Statement* 的执行由于没有标签的 `break` 而突然结束了，那就不采取任何进一步操作，并且 `switch` 语句正常结束。
- 如果 *Statement* 的执行由于任何其他原因而突然结束了，那么 `switch` 语句就出于相同的原因突然结束。由于带标签的 `break` 而导致突然结束的情形，是通过标签语句的一般规则进行处理的。

如同在 C 和 C++ 中一样，开关块中语句的执行“因标签而失败了”。

例如，下面的程序：

```
class Toomany {  
    static void howMany(int k) {  
        switch (k) {  
            case 1: System.out.print("one ");  
            case 2: System.out.print("too ");  
            case 3: System.out.println("many");  
        }  
    }  
    public static void main(String[] args) {  
        howMany(3);  
        howMany(2);  
    }  
}
```

```

        howMany(1);
    }
}

```

包含了一个开关块，块中针对每种情形的代码因失败而进入下一种情形的代码。结果，程序输出：

```

many
too many
one too many

```

如果代码没有按这种方式因失败而从一种情形转到另一种情形，那么应该使用 `break` 语句，如下面的例子所示：

```

class Twomany {
    static void howMany(int k) {
        switch (k) {
            case 1: System.out.println("one");
                    break; // exit the switch
            case 2: System.out.println("two");
                    break; // exit the switch
            case 3: System.out.println("many");
                    break; // not needed, but good style
        }
    }
    public static void main(String[] args) {
        howMany(1);
        howMany(2);
        howMany(3);
    }
}

```

此程序输出：

```

one
two
many

```

## 14.12 while 语句

`while` 语句重复执行 *Expression* 和 *Statement*，直到 *Expression* 的值为 `false`。

**WhileStatement:**

```
while ( Expression ) Statement
```

**WhileStatementNoShortIf:**

```
while ( Expression ) StatementNoShortIf
```

*Expression* 必须具有类型 `boolean` 或 `Boolean`，否则会出生编译时错误。

`while` 语句是通过首先计算 *Expression* 的值来执行的。如果结果的类型是 `Boolean`，

它就遵从拆箱转换（5.1.8 节）。如果 *Expression* 的求值或后续的拆箱转换（如果有的话）出于某个原因突然结束，那么 *while* 语句就出于相同的原因突然结束。否则，执行通过基于结果做出选择而继续。

- 如果值是 *true*，那么就执行包含的 *Statement*。然后从下面进行选择：
  - ◆ 如果 *Statement* 的执行突然结束，那么整个 *while* 语句再次执行，从重新计算 *Expression* 开始。
  - ◆ 如果 *Statement* 的执行突然结束，请参见下面的第 14.12.1 节。
- 如果 *Expression* 的（可能拆箱）值是 *false*，那么就没有采取进一步操作，而且 *while* 语句正常结束。

如果 *Expression* 的（可能拆箱值）在它第一次被计算时是 *false*，那么不执行 *Statement*。

### 14.12.1 突然结束

包含的 *Statement* 的突然结束是以下面的方式进行处理：

- 如果 *Statement* 的执行因为没有标签的 *break* 而突然结束，就不采取任何进一步的操作，并且 *while* 语句正常结束。
  - ◆ 如果 *Statement* 的执行由于没有标签的 *continue* 而突然结束，那么整个 *while* 语句再次执行。
  - ◆ 如果 *Statement* 的执行由于具有标签 *L* 的 *continue* 而突然结束，那么进行下面的选择：
    - ✧ 如果 *while* 语句具有标签 *L*，那么整个 *while* 语句再次被执行。
    - ✧ 如果 *while* 语句没有标签 *L*，那么 *while* 语句由于带标签 *L* 的 *continue* 而突然结束。
- 如果 *Statement* 的执行出于任何其他原因突然结束，那么 *while* 语句就出于相同的原因突然结束。注意，由于带标签的 *break* 而突然结束的情形，是由标签语句的一般规则处理的（14.7 节）。

## 14.13 do 语句

“她不会看到，”他终于简单地说了一句，  
开始时他似乎感觉这句话无需解释。  
——George Eliot, 《Middlemarch》(1871)

*do* 语句重复执行一个 *Statement* 和一个 *Expression*，直到 *Expression* 的值为 *false*。

*DoStatement*:

```
do Statement while ( Expression );
```

*Expression* 具有类型 *boolean* 或 *Boolean*，否则发生一个编译时错误。

do 语句的执行首先执行 *Statement*。然后进行下面的选择：

- 如果 *Statement* 的执行正常结束，那么就计算 *Expression*。如果结果的类型是 `Boolean`，那么它遵从拆箱转换（5.1.8 节）。如果 *Expression* 的求值或后续拆箱转换（如果有的话）出于某个原因突然结束，那么 do 语句就出于相同的原因突然结束。否则，基于结果值有一个选择：
    - ◆ 如果值为 `true`，那么整个 do 语句就再次被执行。
    - ◆ 如果值为 `false`，那么就不采取任何进一步操作，并且 do 语句正常结束。
  - 如果 *Statement* 的执行突然结束，请参阅下面的第 14.13.1 节。
- 执行一个 do 语句始终执行包含的 *Statement* 至少一次。

### 14.13.1 突然结束

包含的 *Statement* 的突然结束是以下面的方式进行处理：

- 如果 *Statement* 的执行由于没有带标签的 `break` 而突然结束，那么就不采取任何的进一步操作，并且 do 语句正常结束。
- 如果 *Statement* 的执行由于没有带标签的 `continue` 而突然结束，那么就计算 *Expression*。然后基于结果值有一个选择：
  - ◆ 若值为 `true`，则整个 do 语句就再执行。
  - ◆ 若值为 `false`，则不采取任何进一步操作，并且 do 语句正常结束。
- 如果 *Statement* 的执行由于带有标签 *L* 的 `continue` 而突然结束，那么进行下面的选择：
  - ◆ 如果 do 语句具有标签 *L*，那么就计算 *Expression*。然后进行下面的选择：
    - ◇ 若 *Expression* 的值是 `true`，则整个 do 语句被再次执行。
    - ◇ 若 *Expression* 的值是 `false`，则不采取任何进一步操作，并且 do 语句正常结束。
  - ◆ 如果 do 语句没有标签 *L*，那么 do 语句由于带标签 *L* 的 `continue` 而突然结束。
- 如果 *Statement* 的执行出于任何其他原因而突然结束，那么 do 语句就出于相同的原因而突然结束。由于带标签的 `break` 而突然结束的情形是通过一般规则（14.7 节）进行处理的。

### 14.13.2 do 语句的例子

下面代码是类 `Integer` 的 `toHexString` 方法的一种可能实现：

```
public static String toHexString(int i) {
    StringBuffer buf = new StringBuffer(8);
    do {
        buf.append(Character.forDigit(i & 0xF, 16));
        i >>= 4;
    } while (i != 0);
    return buf.reverse().toString();
}
```



因为至少必须生成一位，所以 do 语句是一种合适的控制结构。

## 14.14 for 语句

*ForStatement:*

*BasicForStatement*

*EnhancedForStatement*

for 语句有两种形式：

- 基本的 for 语句。
- 增强的 for 语句。

### 14.14.1 语句基础

基本的 for 语句执行一些初始化代码，然后重复执行一个 *Expression*、一个 *Statement* 及一些更新代码，直到 *Expression* 的值为 false。

*BasicForStatement:*

*for* ( *ForInit<sub>opt</sub>* ; *Expression<sub>opt</sub>* ; *ForUpdate<sub>opt</sub>* ) *Statement*

*ForStatementNoShortIf:*

*for* ( *ForInit<sub>opt</sub>* ; *Expression<sub>opt</sub>* ; *ForUpdate<sub>opt</sub>* )

*StatementNoShortIf*

*ForInit:*

*StatementExpressionList*

*LocalVariableDeclaration*

*ForUpdate:*

*StatementExpressionList*

*StatementExpressionList:*

*StatementExpression*

*StatementExpressionList* , *StatementExpression*

*Expression* 的类型必须是 boolean 或 Boolean，否则会发生编译错误。

#### 14.14.1.1 for 语句的初始化

for 语句是先通过执行 *ForInit* 代码来执行的：

- 如果 *ForInit* 代码是语句表达式（14.8 节）的一个列表，那么表达式就按从左到右的顺序进行计算；它们的值（如果有的话）被丢弃。如果任何表达式的求值出于某个原因而突然结束，那么 for 语句就出于相同的原因突然结束；突然结束的语句的右边 *ForInit* 语句没有被求值。

如果 *ForInit* 代码是一个局部变量声明,那么它就像是出现在块中的一个局部变量声明那样(14.4 节)。基本 *for* 语句(14.14 节)的 *ForInit* 部分中声明的一个局部变量的作用域包括下列所有内容:

- 它自己的初始化程序。
- *for* 语句的 *ForInit* 部分中右边的任何进一步声明符。
- *for* 语句的 *Expression* 和 *ForUpdate* 部分。
- 包含的 *Statement*。

如果局部变量声明的执行出于任何原因突然结束,那么 *for* 语句就出于相同的原因突然结束。

- 如果 *ForInit* 不存在,就不采取任何操作。

#### 14.14.1.2 *for* 语句的迭代

接下来, *for* 迭代步骤被执行,如下所述:

- 如果 *Expression* 存在,那就对其进行求值。如果结果的类型是 *Boolean*,那么它就遵从拆箱转换(5.1.8 节)。如果 *Expression* 的求值或后续的拆箱转换(如果有的话)突然结束,那么 *for* 语句就出于相同的原因而结束。否则,那么就基于是否存在 *Expression* 及结果值(如果 *Expression* 存在的话)进行选择:
  - ◆ 若 *Expression* 不存在,或者它存在,并且从它计算(包括任何可能拆箱)中产生的值为 *true*,那么就执行包含的 *Statement*。然后进行如下选择:
    - ◇ 如果 *Statement* 的执行正常结束,那么下面两个步骤就按顺序执行。
      - 首先,如果 *ForUpdate* 部分存在,那么表达式就按从左到右的顺序计算;它们的值(如果有的话)被丢弃。如果任何表达式的求值出于某个原因突然结束,那么 *for* 语句就出于相同的原因突然结束;突然结束的表达式右边的 *ForUpdate* 语句表达式没有被计算。如果 *ForUpdate* 部分不存在,就不采取任何操作。
      - 其次,执行另一个 *for* 迭代步骤。
    - ◇ 如果 *Statement* 的执行突然结束,请参阅下面的第 14.14.1.3 节。
  - ◆ 若 *Expression* 存在,并且从它的计算(包括任何可能的拆箱)产生的值是 *false*,那就不采取任何进一步操作,并且 *for* 语句正常结束。

如果第一次 *Expression* 的(可能拆箱)值在它第一次计算时是 *false*,那么 *Statement* 就没有被计算。

如果 *Expression* 没有存在,那么 *for* 语句可以正常结束的惟一方式是通过使用 *break* 语句。

#### 14.14.1.3 *for* 语句的突然结束

包含的 *Statement* 的突然结束是通过下述方式进行处理:

- 如果 *Statement* 的执行由于没有标签的 *break* 而突然结束,那么就不采取进一步操作,并且 *for* 语句正常结束。
- 如果 *Statement* 的执行因为不带有标签的 *continue* 而突然结束,那么下面两个步

骤按顺序执行：

- ◆ 首先，如果 *ForUpdate* 存在，那么表达式就按左到右的顺序计算；它们的值（如果有的话）被丢弃。如果 *ForUpdate* 部分不存在，就不采取任何操作。
- ◆ 其次，执行另一个 for 迭代。
- 如果 *Statement* 的执行由于带标签 *L* 的 *continue* 而突然结束，那么进行下面的选择：
  - ◆ 若 for 语句具有标签 *L*，那么按顺序执行下面两个步骤：
    - ◇ 首先，如果 *ForUpdate* 部分存在，那么表达式就按从左到右的顺序进行计算，它们的值（如果有的话）被丢弃。如果 *ForUpdate* 不存在，那么就不采取任何操作。
    - ◇ 其次，执行另一个 for 迭代。
  - ◆ 若 for 语句没有标签 *L*，那么 for 语句由于带标签 *L* 的 *continue* 而突然结束。
- 如果 *Statement* 的执行由于任何其他原因而突然结束，那么 for 语句出于相同的原因而结束。注意，由于带标签的 *break* 而突然结束的情形由标签语句（14.7 节）的一般规则处理。

### 14.14.2 增强的 for 语句

增强的 for 语句具有形式：

*EnhancedForStatement*:

*for* ( *VariableModifiers*<sub>opt</sub> *Type Identifier*: *Expression*) *Statement*

*Expression* 必须具有类型 *Iterable*，或者它必须是任何数组类型（10.1 节），否则将发生编译时错误。

在增强的 for 语句（14.14 节）的 *FormalParameter* 部分中声明的局部变量的作用域是包含的 *Statement*。

增强 for 语句的含义是通过转换到基本的 for 语句而给出的。

如果 *Expression* 的类型是 *Iterable* 的子类型，那么令 *I* 是表达式 *Expression.iterator()* 的类型。那么增强的 for 语句就等价于形如下面的基本 for 语句：

```
for (I #i = Expression.iterator(); #i.hasNext(); ) {
    VariableModifiersopt Type Identifier = #i.next();
    Statement
}
```

其中 # 是一个编译生成的标识符，该标识符不同于在增强 for 语句出现的点的作用域中（6.3 节）的任何其他标识符（编译器生成或其他的什么）。

否则，*Expression* 必须有一个数组类型 *T[]*。令 *L*<sub>1</sub>...*L*<sub>*m*</sub> 为增强的 for 语句正前面的（可能为空的）标签序列。那么增强 for 语句的含义是通过下面的基本 for 语句给出的：

```
T[] a = Expression;
L1: L2: ... Lm:
```

```

for (int i = 0; i < a.length; i++) {
    VariableModifiersopt Type Identifier = a[i];
    Statement
}

```

其中 *a* 和 *i* 是编译器生成的标识符, 该标识符不同于增强 for 语句出现所在的点的作用域中的任何其他标识符 (编译器生成的或其他的什么)。

### 讨论

下面的例子 (计算一个整数数组的和) 展示了增强的 for 对于数组是如何工作的:

```

int sum(int[] a) {
    int sum = 0;
    for (int i : a)
        sum += i;
    return sum;
}

```

下面的例子将增强的 for 语句与自动拆箱结合起来, 以便将柱状图转换成一个频率表:

```

Map<String, Integer> histogram = ...;
double total = 0;
for (int i : histogram.values())
    total += i;
for (Map.Entry<String, Integer> e : histogram.entrySet())
    System.out.println(e.getKey() + " " + e.getValue() / total);

```

## 14.15 break 语句

break 语句将控制传送出封闭的语句。

*BreakStatement:*

```
break Identifieropt;
```

带有标签的 break 语句尝试将控制传送到直接封闭的方法或初始化程序块的最内部的封闭的 switch、while、do 或 for 语句; 然后此语句 (称为中断目标) 立即正常结束。

更准确地说, 不带标签的 break 语句始终突然结束, 原因是 break 没有带标签。如果在直接封闭的方法中没有 switch、while、do 或 for 语句, 那么构造函数或初始化程序就封闭 break 语句, 生成一个编译时错误。

带有标签 *Identifier* 的 break 语句尝试将控制传送给封闭的标签语句 (14.7 节), 该语句具有与它的标签相同的 *Identifier*; 然后此语句 (称为中断目标), 立即正常结束。在这种情形中, break 目标需要是一个 while、do、for 或 switch 语句。break 语句必须引用直接封闭方法或初始化程序块中的一个标签。没有非局部的跳转。

更准确地讲, 带有标签 *Identifier* 的 break 语句始终突然结束, 原因是带有标签 *Identifier* 的一个 break。如果没有将 *Identifier* 作为其标签的标签语句封闭 break 语句,

那么就发生编译时错误。

然后可以看到 `break` 语句始终突然结束。

前面描述的是“尝试传送控制”而不是只“传送控制”，因为如果中断目标中有任何 `try` 语句（14.20 节），其 `try` 块包含 `break` 语句，那么这些 `try` 语句的任何 `finally` 子句就被执行，执行顺序是从最内部到最外部，然后将控制传送给中断目标。`finally` 子句的突然结束可能中断 `break` 语句发出的控制传送。

在下面的例子中，算术图是由一组数组表示的。该图包括一组节点和一组边：每条边是一个箭头，该箭头从某个点指向某个其他点，或从一个点指向其自身。在本例中，假定没有冗余的边：也就是说，对于任何两个节点  $P$  和  $Q$ ，其中  $Q$  可能是与  $P$  相同的，那么就至少有一条从  $P$  到  $Q$  的边。节点是由整数表示的，对于每个  $i$  和  $j$ ，有一条从节点  $i$  到节点 `edges[i][j]` 的边，并且数组引用 `edges[i][j]` 没有抛出一个 `IndexOutOfBoundsException`。

在给定整数  $i$  和  $j$  的情况下，方法 `loseEdges` 的任务是要构造一个新图，方式是复制给定的图，但省去从节点  $i$  到节点  $j$  的边（如果有的话），及从节点  $j$  到节点  $i$  的边（如果有的话）：

```
class Graph {
    int edges[][];
    public Graph(int[][] edges) { this.edges = edges;
}

    public Graph loseEdges(int i, int j) {
        int n = edges.length;
        int[][] newedges = new int[n][];
        for (int k = 0; k < n; ++k) {
            edgelist: {
                int z;
                search: {
                    if (k == i) {
                        for (z = 0; z < edges[k].length; ++z)
                            if (edges[k][z] == j)
                                break search;
                    } else if (k == j) {
                        for (z = 0; z < edges[k].length; ++z)
                            if (edges[k][z] == i)
                                break search;
                    }
                    // No edge to be deleted; share this list.
                    newedges[k] = edges[k];
                    break edgelist;
                } //search
                // Copy the list, omitting the edge at position z.
                int m = edges[k].length - 1;
                int ne[] = new int[m];
                System.arraycopy(edges[k], 0, ne, 0, z);
            }
        }
    }
}
```

```

        System.arraycopy(edges[k], z+1, ne, z, m-z);
        newedges[k] = ne;
    } //edgelist
}
return new Graph(newedges);
}
}

```

注意两个语句标签 `edgelist` 和 `search` 的使用以及 `break` 语句的使用。这允许复制清单（省去一条边）的代码，以便在两个不同的测试——从节点  $i$  到节点  $j$  的边的测试和从节点  $j$  到节点  $i$  的边测试——之间进行共享。

## 14.16 continue 语句

福尔摩斯先生在他的委托人停了一下、使劲地吸了一下鼻烟以便稍加思索的时候说，  
 “你的这段经历真是最有趣不过了。请你继续讲你的这段十分有趣的事吧。”  
 ——亚瑟·柯南·道尔，《红发会》（1891）

`continue` 语句只可以出现在 `while`、`do` 或 `for` 语句中；这 3 种语句被称为迭代语句。控制传递给迭代语句的循环继续点。

*ContinueStatement:*

```
continue Identifieropt;
```

没有标签的 `continue` 语句尝试将控制传送给直接封闭方法或初始化程序块的最内部封闭语句 `while`、`do` 或 `for`；然后，此语句（称为继续目标）立即结束当前的迭代并开始一个新的迭代。

准确地讲，这样的 `continue` 语句始终突然结束，原因是 `continue` 没有标签。如果没有直接封闭方法或初始化程序块的 `while`、`do` 或 `for` 语句封闭 `continue` 语句，则发生一个编译时错误。

带标签 *Identifier* 的 `continue` 语句尝试将控制传送给封闭的标签语句（14.7 节），该语句和它的标签具有相同的 *Identifier*；然后此语句（称为继续目标）立即结束当前迭代，并开始一个新的迭代。继续目标必须是一个 `while`、`do` 或 `for` 语句，否则会发生编译时错误。`continue` 语句必须引用直接封闭方法或初始化程序块的一个标签。没有非局部跳转。

更准确地说，具有 *Identifier* 的 `continue` 语句始终突然结束，原因是 `continue` 具有标签 *Identifier*。如果没有将 *Identifier* 作为其标签的标签语句包含 `continue` 语句，将发生一个编译时错误。

然后可以看到 `continue` 语句总是突然结束。

关于处理由 `continue` 而引起的突然结束的讨论，请参阅对 `while` 语句（14.12 节）、`do` 语句（14.13 节）和 `for` 语句（14.14 节）的描述。

前面描述的是“尝试传送控制”而不是只“传送控制”，因为如果继续目标中如何有任



何 try 语句(14.20 节),其 try 块包含 continue 语句,那么这些 try 块的任何 finally 子句就被执行,执行顺序是从最内部到最外部,然后将控制传送给继续目标。finally 子句的突然结束可能中断由 continue 发起的控制传送。

在前一节的 Graph 示例中, break 语句之一用于完成 for 循环最外部的整个体的执行。如果 for 循环本身有标签的话,此 break 就可以用 continue 替换:

```
class Graph {
    . . .
    public Graph loseEdges(int i, int j) {
        int n = edges.length;
        int[][] newedges = new int[n][];
        edgelists: for (int k = 0; k < n; ++k) {
            int z;
            search: {
                if (k == i) {
                    . . .
                } else if (k == j) {
                    . . .
                }
                newedges[k] = edges[k];
                continue edgelists;
            } // search
            . . .
        } // edgelists
        return new Graph(newedges);
    }
}
```

要使用哪个语句(如果有一种可选的话)是编程风格的事情。

## 14.17 return 语句

“知道吗,至高无上的神和 Helium 的人民,”他说,“John Carter,  
曾经的 Helium 的王子,声称自己刚刚从 Valley Dor 回来……”  
——Edgar Rice Burroughs,《The Gods of Mars》(1913)

return 语句将控制返回给方法(8.4 节、15.12 节)或构造函数(8.8 节、15.9 节)的调用程序。

**ReturnStatement:**

return *Expression<sub>opt</sub>*;

没有 *Expression* 的 return 语句必须包含在使用关键字 void 声明的方法体中,以避免返回任何值(8.4 节),或者必须包含在构造函数(8.8 节)的体中。如果 return 语句出现在实例初始化程序或静态初始化程序(8.7 节)中,那么就发生一个编译时错误。没有 *Expression* 的 return 语句尝试将控制传送给包含它的方法或构造函数的调用程序。

更准确地讲,没有 *Expression* 的 *return* 语句总是突然结束,原因是没有值的 *return*。

具有 *Expression* 的 *return* 语句必须包含在被声明以返回一个值 (8.4 节) 的方法声明中,否则会发生编译时错误。*Expression* 必须表示相同类型 *T* 的一个变量或值,否则会发生编译时错误。类型 *T* 必须可赋值 (5.2 节) 给方法的声明结果类型,否则发生一个编译时错误。

具有 *Expression* 的 *return* 语句尝试将控制传送给包含它的方法的调用程序;*Expression* 的值变成了方法调用的值。更准确地讲,这样的 *return* 语句的执行首先计算 *Expression*。如果 *Expression* 的计算出于某个原因突然结束,那么 *return* 语句就出于该原因突然结束。如果 *Expression* 的求值正常结束,产生一个值 *V*,那么 *return* 语句就突然结束,原因是 *return* 具有值 *V*。如果表达式的类型是 *float*,并且不是精确浮点的 (15.4 节),那么值可能是浮点值集合或浮点扩展指数值集合 (4.2.3 节) 的一个元素。如果表达式的类型是 *double*,并且不是精确浮点的,那么值可能是 *double* 值集合或双精度扩展指数值集合的一个元素。

可以看出, *return* 语句始终突然结束。

前面描述的是“尝试传送控制”而不是只“传送控制”,因为如果方法或构造函数中有任何 *try* 语句 (14.20 节),其 *try* 块包含 *return* 语句,那么将执行这些 *try* 语句的任何 *finally* 子句,执行顺序是从最内部到最外部,然后将控制传送给方法或构造函数的调用程序。*finally* 子句的突然结束可以中断 *return* 语句发起的控制传送。

## 14.18 throw 语句

*throw* 语句导致抛出一个异常 (第 11 章) 结果是控制 (11.3 节) 的直接传送,这可能退出多条语句和多个构造函数、实例初始化程序、静态初始化程序和字段初始化程序计算及方法调用,直到找到一个 *try* 语句 (14.20 节) 来捕获异常。如果没有找到这样的 *try* 语句,那么执行 *throw* 的线程的执行 (第 17 章) 就在线程所属的线程组 *uncaughtException* 方法的调用后终止 (11.3 节)。

*ThrowStatement*:

```
throw Expression;
```

抛出语句可以抛出类型为 *E* 的异常,前提是抛出表达式的静态类型是 *E* 或 *E* 的一个子类型,否则抛出表达式可能抛出 *E*。

抛出语句中的 *Expression* 必须表示引用类型的一个引用变量或值,该引用类型可赋值给类型 *Throwable*,否则将发生编译时错误。而且,至少要有下面的 3 个条件之一为假,否则会发生编译时错误:

- 异常不是一个检查异常 (11.2 节) —— 特别地,下面的情形之一为真:
  - ◆ *Expression* 的类型是类 *RuntimeException* 或 *RuntimeException* 的一个子类。
  - ◆ *Expression* 的类型是类 *Error* 或 *Error* 的一个子类。
- *throw* 包含在 *try* 语句 (14.20 节) 的 *try* 块中,并且 *Expression* 类型可赋予 (5.2

节) try 语句的至少一个 catch 语句的参数的类型 (在本例中, 我们说抛出值是由 try 语句捕获的)。

- throw 语句包含在一个方法或构造函数声明中, 并且 *Expression* 的类型可赋予 (5.2 节) 声明中的 throws 子句 (8.4.6 节、8.8.5 节) 列出的至少一种类型。

throw 语句首先计算 *Expression*。如果 *Expression* 的求值出于某个原因突然结束, 那么 throw 就出于同样的原因突然结束。如果 *Expression* 的求值突然结束, 产生一个非 null 值, 那么 throw 语句就突然结束, 原因是具有值 *V* 的 throw。如果 *Expression* 的求值正常结束, 产生一个 null 值, 那么就创建并抛出类 NullPointerException 的实例 *V'*, 而非 null。然后 throw 语句突然结束, 原因是具有值 *V'* 的 throw。

可以看出, throw 语句始终正常结束。

如果没有任何封闭的 try 语句 (14.20 节), 其 try 块包含 throw 语句, 那么这些 try 语句的任何 finally 子句将在控制向外传送时被执行, 直到抛出值被捕获。注意, finally 子句的异常结束可以中断 throw 语句发出的控制传送。

如果一条 throw 语句包含在方法声明中, 但其值没有被包含它的某个 try 语句捕获, 那么方法的调用就由于 throw 而突然结束。

如果 throw 语句包含在一个构造函数声明中, 但其值没有被某些包含它的 try 语句捕获, 那么调用构造函数的类实例创建表达式将由于 throw 而突然结束。

如果 throw 语句包含在一个静态初始化程序 (8.7 节), 那么编译时检查确保了它的值或者始终是一个未检查异常, 或者始终由包含它的 try 语句捕获。如果在运行时 (不管该检查), 该值没有被包含 throw 语句的某个 try 语句捕获, 那么该值就重新抛出, 前提是它是类 Error 的一个实例或它的子类之一; 否则, 它被包装在 ExceptionInitializerError 对象, 然后该对象被抛出 (12.4.2 节)。

如果 throw 语句包含在实例初始化程序 (8.6 节) 中, 那么编译时检查确保它的值或者始终是一个未异常, 或者始终被包含它的某个 try 语句捕获, 否则抛出异常的类型 (或它的超类之一) 在每个类的构造函数的 throws 子句中发生。

按约定, 用户声明的可抛出类型通常应该被声明成类 Exception 的子类, Exception 是类 Throwable 的一个子类 (11.5 节)。

## 14.19 synchronized 语句

synchronized 语句代表执行中的线程获得互斥锁 (17.1 节), 执行一个块, 然后释放锁。当执行线程拥有块时, 就没有其他线程可以获得锁。

*SynchronizedStatement:*

synchronized ( *Expression* ) *Block*

*Expression* 的类型必须是一个引用类型, 否则会发生编译时错误。

synchronized 语句是通过首先计算 *Expression* 执行的。

如果 *Expression* 的求值出于某个原因突然结束, 那么 synchronized 语句就出于相

同的原因突然结束。

否则, 如果 *Expression* 的值是 `null`, 那么就抛出 `NullPointerException`。

否则, 令 *Expression* 的非 `null` 值为 *V*。执行线程锁住和 *V* 相关的锁, 然后 *Block* 被执行。如果 *Block* 的执行突然结束, 那么锁就被解开, 并且 `synchronized` 语句突然结束。如果 *Block* 的执行出于任何原因突然结束, 那么锁就被解开, 然后 `synchronized` 语句出于相同的原因突然结束。

获取与对象有关的锁本身不防止其他线程访问对象的字段, 或者调用对象上的非同步化方法。其他线程也可以以约定的方式使用 `synchronized` 方法或 `synchronized` 语句以取得互斥的效果。

`synchronized` 语句获得的锁与通过 `synchronized` 方法隐式获得的锁是相同的; 参阅第 8.4.3.6 节。一个简单的线程可以多次持有一个锁。

下面的例子:

```
class Test {
    public static void main(String[] args) {
        Test t = new Test();
        synchronized(t) {
            synchronized(t) {
                System.out.println("made it!");
            }
        }
    }
}
```

输出如下:

```
made it!
```

如果不允许单个线程对一个锁多次加锁, 那么本示例将发生死锁。

## 14.20 try 语句

此乃我辈心灵饱受考验的时代。

——托马斯·潘恩,《北美的危机》(1780)

……他们全都开始想尽一切办法捕猎, 直到用完了他们靴子跟部的黑色火药。

——Samuel Foote (18 世纪美国作家)

`try` 语句执行一个块。如果值被抛出, 并且 `try` 语句有可以捕获它的一个或多个 `catch` 子句, 则将把控制传送给第一个这样的 `catch` 子句。如果 `try` 语句具有 `finally` 子句, 那么就执行代码的另一个块, 而不管 `try` 块是正常或突然结束, 并且不管 `catch` 子句是否是第一个给定控制。

*TryStatement:*

```
try Block Catches
try Block Catchesopt Finally
```

*Catches:*

```
CatchClause
Catches CatchClause
```

*CatchClause:*

```
catch ( FormalParameter ) Block
```

*Finally:*

```
finally Block
```

下面重复了第 8.4.1 节，使得陈述更加清楚：

*FormalParameter:*

```
VariableModifiers Type VariableDeclaratorId
```

下面重复了第 8.3 节，以使得陈述更加清楚：

*VariableDeclaratorId:*

```
Identifier
VariableDeclaratorId [ ]
```

紧跟在关键字 `try` 后面的 *Block* 称为 `try` 语句的 `try` 块。紧跟在关键字 `finally` 后面的 *Block* 语句称为 `try` 语句的 `finally` 块。

一个 `try` 语句可以有几个 `catch` 语句（也称为异常处理程序）。一个 `catch` 语句必须正好有一个参数（称为异常参数）；异常参数的声明类型必须是类 `Throwable` 或 `Throwable` 的一个子类（而不仅是一个子类型），否则发生编译时错误。尤其是当异常参数的声明类型是一个类型变量（4.4 节）时，它就是一个编译时错误。参数变量的作用域是 `catch` 子句的 *Block*。

捕获子句的异常参数一定不能跟直接封闭在 `catch` 子句中的方法或初始化程序块的局部变量或参数有相同的名称，否则将发生编译时错误。

在 `try` 语句（14.20 节）的 `catch` 子句中声明的异常处理程序的参数的作用域，是与 `catch` 有关的整个块。在 `catch` 子句的 *Block* 中，参数的名称可以重新声明为直接封闭方法或初始化程序块的局部变量，它不可以重新声明为直接封闭方法或初始化程序块的 `try` 语句中捕获语句的一个异常参数，否则发生编译时错误。但异常参数可以被在嵌套于 `catch` 子句的 *Block* 中的类声明内部的任何地方屏蔽（6.3.1 节）。

`try` 语句可以抛出异常类型 *E*，当且仅当下列之一成立：

- `try` 块可以抛出 *E*，*E* 没有赋予 `try` 语句的任何 `catch` 参数，并且没有 `finally` 块存在，或者 `finally` 块可以正常结束。
- 或者 `try` 语句的某个 `catch` 块可以抛出 *E*，并且没有 `finally` 存在，或者 `finally` 块可以正常结束。



- 或者 `finally` 存在，并且可以抛出  $E$ 。

如果声明为 `final` 的异常参数被赋予到捕获子句的体中，那么它就是一个编译时错误。

如果 `catch` 子句捕获检查异常类型  $E1$ ，但不存在检查异常类型  $E2$ ，使得下面所列都成立，那么它就是一个运行时异常：

- $E2 <: E1$ 。
- 对应于 `catch` 子句的 `try` 块可以抛出  $E2$ 。
- 没有直接封闭的 `try` 语句的前面的 `catch` 块捕获  $E2$  或  $E2$  的超类型。

除非  $E$  是类 `Exception`。

异常参数不能使用限定名称（6.6 节）进行引用，只能通过简单名称进行引用。

异常处理程序按从左到右的顺序进行考虑：最早的可能 `catch` 子句接受异常，将抛出的异常对象接收为它的实际参数。

`finally` 块子句确保 `finally` 块在 `try` 块及可能执行的任何 `catch` 块后执行，而不管控制如何离开 `try` 块或 `catch` 块。

`finally` 块的处理是相当复杂的，因此具有和不具有 `finally` 块的两种情形需要分别进行描述。

### 14.20.1 try-catch 的执行

我们至高无上的任务是恢复我们向前的正常路线。

——摘自美国第 29 任总统 Warren G. Harding 的就职演说（1921）

不具有 `finally` 块的 `try` 语句是首先通过执行 `try` 块而执行的。然后进行下列选择：

- 如果 `try` 块的执行正常结束，那么就不采取任何进一步操作，并且 `try` 块正常结束。
- 如果 `try` 块的执行由于抛出值  $v$  而突然结束，那么进行下列选择：
  - ◆ 若  $v$  的运行时类型可赋予（5.2 节）`try` 语句的任何 `catch` 子句的 *Parameter*，那么第一个（最左边）这样的 `catch` 子句被选择。值  $v$  被赋予选定的 `catch` 子句的参数，并且执行该 `catch` 子句的 `Block`。如果那一块正常结束，那么 `try` 语句正常结束；如果那一块出于任何原因而突然结束，那么 `try` 语句出于相同的原因突然结束。
  - ◆ 若  $v$  的运行时类型没有赋予 `try` 语句的任何 `catch` 子句，那么 `try` 语句就由于抛出值  $v$  而突然结束。
- 如果 `try` 块的执行出于任何其他原因而突然结束，那么 `try` 语句就出于相同的原因突然结束。

在下面的例子中：

```
class BlewIt extends Exception {
    BlewIt() { }
    BlewIt(String s) { super(s); }
}
class Test {
```



```

static void blowUp() throws BlewIt { throw new BlewIt(); }
public static void main(String[] args) {
    try {
        blowUp();
    } catch (RuntimeException r) {
        System.out.println("RuntimeException:" + r);
    } catch (BlewIt b) {
        System.out.println("BlewIt");
    }
}
}

```

异常 `BlewIt` 通过方法 `blowUp` 抛出。`main` 的体中的 `try-catch` 语句具有两个 `catch` 子句。异常的运行时类型是 `BlewIt`，它不能赋给类型为 `RuntimeException` 的变量，但可以赋给类型为 `BlewIt` 的变量，因此此例子的输出是：

`BlewIt`

### 14.20.2 try-catch-finally 的执行

在伟大的船长和工程师完成了他们的工作之后，  
 在高尚的发明家之后，  
 在科学家、化学家、地质学家和人类文化学者之后，  
 最后将出现诗人……  
 ——沃尔特·惠特曼，《向着印度前进》（1870）

具有 `finally` 块的 `try` 语句是首先通过执行 `try` 块来执行的。然后进行下列选择：

- 如果 `try` 块的执行突然结束，那么 `finally` 块就被执行，然后进行下面的选择：
  - ◆ 若 `finally` 块突然结束，则 `try` 语句就正常结束。
  - ◆ 若 `finally` 块出于原因 `S` 而突然结束，那么 `try` 语句就出于原因 `S` 突然结束。
- 如果 `try` 块的执行由于抛出值 `v` 而突然结束，那么就进行下面的选择：
  - ◆ 若 `v` 的运行时类型赋予 `try` 语句的任何 `catch` 语句的参数，那么第一个（最左边）这样的 `catch` 子句被选择。值 `v` 赋予选定 `catch` 子句的参数，并执行那个 `catch` 子句中的 `Block`。然后进行下面的选择：
    - ◇ 如果 `catch` 块正常结束，那么 `finally` 块就被执行。然后进行下面的选择：
      - 若 `finally` 块正常结束，那么 `try` 语句正常结束。
      - 若 `finally` 块出于任何原因而突然结束，那么 `try` 语句出于相同的原因而突然结束。
    - ◇ 如果 `catch` 块出于原因 `R` 突然结束，那么 `finally` 块被执行。然后进行下面的选择：
      - 若 `finally` 块正常结束，那么 `try` 语句出于原因 `R` 而突然结束。
      - 若 `finally` 块出于原因 `S` 突然结束，那么 `try` 语句出于原因 `S` 突然结束，并

且原因  $R$  被丢弃)。

- ◆ 若  $V$  的运行时类型没有赋予 try 语句的任何 catch 子句的参数, 那么就执行 finally 块。然后进行下面的选择:
  - ✧ 如果 finally 块正常结束, 那么 try 语句就由于抛出值  $V$  而突然结束。
  - ✧ 如果 finally 块出于原因  $S$  而突然结束, 那么 try 语句就出于原因  $S$  而突然结束 (并且值  $V$  的抛出被丢弃和忘记)。
- 如果 try 块的执行出于任何其他原因  $R$  而突然结束, 那么 finally 块就被执行。然后进行下面的选择:
  - ◆ 若 finally 块突然结束, 那么 try 语句就出于原因  $R$  而突然结束。
  - ◆ 若 finally 块出于原因  $S$  而突然结束, 那么 try 语句就出于原因  $S$  而突然结束 (并且原因  $R$  被丢弃)。

请看下面的例子:

```
class BlewIt extends Exception {
    BlewIt() { }
    BlewIt(String s) { super(s); }
}

class Test {
    static void blowUp() throws BlewIt {
        throw new NullPointerException();
    }
    public static void main(String[] args) {
        try {
            blowUp();
        } catch (BlewIt b) {
            System.out.println("BlewIt");
        } finally {
            System.out.println("Uncaught Exception");
        }
    }
}
```

输出如下:

```
Uncaught Exception
java.lang.NullPointerException
    at Test.blowUp(Test.java:7)
    at Test.main(Test.java:11)
```

由方法 blowUp 抛出的 NullPointerException (是 RuntimeException 的一个类别) 不是由 main 中的 try 语句捕获的, 因为 NullPointerException 不可赋予类型为 BlewIt 的变量。这导致了 finally 子句执行, 其后线程执行 main (是测试程序的一个仅有线程) 由于未捕获的异常而终止了, 未捕获的异常导致了输出异常名称和一个简单的回溯。但回溯不是本规范所需的。

## 讨论

与强制回溯有关的问题是异常可以在程序中的一个点创建,并在较后面的一个点抛出。在异常中存储回栈溯堆绝对是昂贵的,除非它真的被抛出了(在这种情形中可能生成跟踪,同时展开堆栈)。因此,我们不强制在每个异常中进行堆栈跟踪。

## 14.21 不可到达语句

那看上去像一条路。

这条路是从这里通往山顶吗?

——罗伯特·弗罗斯特,《山》(1915)

如果语句因为它是不可到达的而不能执行,那么它就是一个编译时错误。每个 Java 编译器必须执行这里指定的保守流分析,以确保所有的语句是可到达的。

本节致力于术语“可到达的”的精确解释。想法是必须有从包含语句的构造函数、方法、实例初始化方法或静态初始化的方法开始到语句本身的可能执行路径。该分析考虑了语句的结构。除了其条件表达式具有常量值 true 的 while、do 和 for 语句的特殊处理,表达式的值在流分析中没有进行考虑。

例如,Java 编译器将接受代码:

```
{
    int n=5;
    while (n>7) k=2;
}
```

即使  $n$  的值在编译时是已知的,并且原则上它在编译时可以知道赋值给  $k$  可能从未被执行。

Java 编译器必须根据本列提出的规则进行操作。

本节中的规则定义两个技术术语:

- 语句是否是可达的。
- 语句是否可以正常结束。

这里的定义允许语句正常结束,只要它是可达的。

规则如下:

- 构造函数、方法、初始化方法或静态初始化程序的体的块是可达的。
- 当且仅当不是开关块的空块是可达的时,它可能正常结束。只要不是开关块的非空块当中的最后一条语句正常结束,该块就可以正常结束。不是开关块的非空块中第一条语句是可达的,当且仅当该块是可达的。不是开关块的空块中的每两个语句  $s$  是可达的,当且仅当  $s$  前面的语句可以正常结束。
- 本地类声明语句可以突然结束,当且仅当它是可达的。
- 局部变量声明语句可能正常结束,当且仅当它是可达的。

- 空语句可能正常结束，当且仅当它是可到达的。
- 标签语句可能正常结束，如果至少下面之一成立的话：
  - ◆ 包含的语句可以正常结束。
  - ◆ 有一个退出标签语句的可到达 break 语句。

当且仅当标签语句是可到达的时候，包含的语句是可到达的。

- 表达式语句可以正常结束，当且仅当它是可到达的。
- 以非一般的方式处理 if 语句，不管它是否有 else 部分。因此，在本节的后面它被分开讨论了。
- assert 语句可以正常结束，当且仅当它是可到达的。
- 当且仅当下面之一为真时，switch 语句就可以正常结束：
  - ◆ 开关块中最后一个语句可以正常结束。
  - ◆ 开关块是空的或只包含开关标签。
  - ◆ 在开关块语句组后面至少有一个开关标签。
  - ◆ 开关块没有包含 default 标签。
  - ◆ 有一个退出 switch 语句的可到达 break 语句。
- 当且仅当开关块的 switch 语句是可到达的，开关块才是可到达的。
- 当且仅当开关块的 switch 语句是可到达的，并且至少下面之一为真时，开关块中的该语句才是可到达的：
  - ◆ 它带有 case 或 default 标签。
  - ◆ switch 块中它前面有一个语句，并且该语句可以正常结束。
- while 语句可以正常结束，当且仅当下面之一为真时：
  - ◆ while 语句是可到达的，并且条件表达式不是带有值 true 的常量表达式。
  - ◆ 有一个退出 while 语句的可到达 break 语句。

当且仅当 while 语句是可到达的，并且条件表达式不是其值为 false 的常量表达式时，包含的语句是可到达的。

- do 语句可以正常结束，当且仅当下面之一为真时：
  - ◆ 包含的语句可以正常结束，并且条件表达式不是带有值 true 的常量表达式。
  - ◆ do 语句包含不带标签的可到达 continue 语句，并且 do 语句是最内部的 while、do、for 语句，这些语句包含了 continue 语句，并且条件表达式不是带有值 true 的常量表达式。
  - ◆ do 语句包含带有标签 L 的可到达 continue 语句，并且 do 语句具有标签，条件表达式不是带有值 true 的常量表达式。
  - ◆ 有一个退出 do 语句的可到达 break 语句。

当且仅当 do 语句是可到达时，包含的语句是可到达的。

- 当且仅当下面之一为真时，基本的 for 语句可以正常结束：
  - ◆ for 语句是可到达的，没有条件表达式，并且条件表达式不是一个带有值 true 的常量表达式。
  - ◆ 有一个退出 for 语句的可到达 break 语句。

当且仅当 for 语句是可到达的，并且条件表达式不是一个其值为 false 的常量表达式时，包含的语句是可到达的。

- 当且仅当增强的 for 语句是可到达的，该语句就可以正常结束。
- break、continue、return 或 throw 语句不可以正常结束。
- synchronized 语句可以正常结束，当且仅当包含的语句可以正常结束的话。当且仅当 synchronized 语句是可到达的话，包含的语句就是可到达的。
- 当且仅当下面两者都为真的话，try 语句可以正常结束。
  - ◆ try 块或以正常结束，或者任何 catch 块可以正常结束。
  - ◆ 如果 try 块有一个 finally 块，那么 finally 块可以正常结束。
- 当且仅当 try 语句是可到达的，try 块就是可到达的。
- 当且仅当下面两者都为真时，catch 块 C 是可到达的：
  - ◆ try 块是一些表达式或 throw 语句是可到达的，并且可以抛出一个异常，其类型可赋予 catch 子句 C 的参数。（当且仅当包含它的最内部语句是可到达时，表达式才被认为是可到达的。）
  - ◆ 在 try 语句中没有较前面的 catch 块 A，使得 C 的参数的类型与 A 的参数的类型相同，或是它的一个子类。
- 如果 finally 块存在的话，那么当且仅当 try 语句是可到达的话，它就是可到达的。

我们可能希望 if 语句按下面的方式进行处理，但 Java 编程语言实际上没有使用什么规则：

- HYPOTHETICAL：当且仅当下面之一为真时，if-then 语句可以正常结束：
  - ◆ if-then 语句是可到达的，并且条件表达式不是其值为 true 的常量表达式。
  - ◆ then-语句可以正常结束。
- 当且仅当 if-then 语句是可到达的，并且条件表达式不是其值为 false 的常量表达式时，then-语句是可到达的。
- HYPOTHETICAL：当且仅当 then-语句可以正常结束，或者 else-语句可以正常结束时，那么 if-then-else 语句可以正常结束。当且仅当 if-then-else 语句是可到达的，并且条件表达式不是一个其值为 false 的常量表达式，那么 then-语句是可到达的。当且仅当 if-then-else 语句是可到达的，并且条件表达式不是一个其值为 true 的常量表达式时，那么 else 语句是可到达的。

这种方法与其他控制结构的处理一致。但为了允许 if 语句方便地用于“条件编译”目的，实际的规则是不同的。

if 语句的实际规则如下：

- ACTUAL：当且仅当 if-then 语句是可到达的，那么该语句就可以正常结束。当且仅当 if-then 语句是可到达的，then-语句是可到达的。
- ACTUAL：当且仅当 then-语句可以正常结束，或者 else-语句可以正常结束时，if-then-else 语句可以正常结束。当且仅当 if-then-else 语句是可到达的，then-语句才是可到达的。当且仅当 if-then-else 语句是可到达的，else-语句才是可到达的。

再如，下面的语句导致了一个编译时错误：

```
while (false) { x=3; }
```

因此语句 `x=3;` 是不可到达的；但相似的例子：

```
if (false) { x=3; }
```

没有导致编译时错误。一个优化的编译器可能认识到语句 `x=3;` 从未被执行，因而可能选择从生成的 class 文件中忽略语句的代码，但从这里指定的技术意义而言，语句 `x=3;` 不被认为是“不可到达的”。

这种不同的处理的基本原理是要允许程序员定义如下的“标志变量”：

```
static final boolean DEBUG = false;
```

然后编写如下的代码：

```
if (DEBUG) { x=3; }
```

这种想法是应该可以将 `DEBUG` 的值从 `false` 更改到 `true`，或从 `true` 更改到 `false`，然后在对程序文本没有做任何其他更改时正确地编译代码。

这种“条件编译”能力对二进制兼容性（第 13 章）有很大的影响，并与二进制兼容性也有关系。如果使用这样的“标志”变量的一组类被编译，并且忽略条件代码，以后它就不能够只分发包含标志定义的类或接口的一个新版本。因此，对标志值的更改与前面存在的二进制（13.4.9 节）不是二进制兼容的（还有这样的不兼容性的其他原因，比如 `switch` 语句中的 `case` 标签中常量的使用；参阅第 13.4.9 节）。

只要理清头绪，您的工作不应该混乱……  
——罗伯特·弗罗斯特，《The Generations of Men》（1914）



## 表 达 式

当你明白你正在说什么，并用文字表达它时，你就知道一些关于它的事情；但是，当你不明白它时，当你不能用文字表达它时，你对它的了解就属于缺乏和不满意那一类：这可能就是知识的起源，但是，你几乎无法在你的思想中推进到科学的阶段。

——William Thompson, Lord Kelvin（开尔文温标的提出者）

程序中的大多数工作通过对表达式求值完成，要么利用表达式的副作用，比如对变量的赋值，要么利用表达式的值，这些值可以用作较大表达式中的参数或操作数，或者这些值会影响语句的执行顺序，也可以两者同时利用。

本章描述了表达式的含义及它们的求值规则。

## 15.1 计算、表示和结果

当对程序中的表达式进行求值（执行）时，结果表示为以下三种方式之一：

- 变量（4.12 节）（在 C 中，这称为 lvalue）
- 值（4.2 节，4.3 节）
- 什么也没有（也就是说表达式为 void）

表达式的求值也可能产生副作用，因为表达式可能包含嵌入的赋值、增量运算符、减量运算符和方法调用。

当且仅当表达式是一个调用没有返回值的方法（15.12 节）（也就是声明为 void 的方法）的方法调用时，表达式不表示什么（8.4 节）。这样的表达式只可以用作表达式语句（14.8 节），因为表达式可以出现的每两个上下文要求表达式表示某些东西。是方法调用的表达式语句也可以调用产生结果的方法；在这种情况下，方法返回的值被悄悄地丢弃。

值集合转换（5.1.13 节）被应用到产生值的每个表达式的结果。

每个表达式在下面的两种情况下发生：

- 某个（类或接口）类型的声明在下面某一个当中被声明：字段初始化程序，静态初始化程序，实例初始化程序，构造函数声明，注释，或者方法的代码。
- 数据包或顶层类型声明的注释。

## 15.2 变量作为值

如果表达式表示一个变量，并且需要一个值以便用在进一步求值中，那么就会使用该变量的值。在此上下文中，如果表达式表示一个变量或一个值，我们就可以简单地说出表达式的值。

如果类型 `float` 或 `double` 的变量的值以这种方式进行使用，那么值集合转换(5.1.13 节)就被应用到变量的值。

## 15.3 表达式的类型

如果表达式表示一个变量或一个值，那么表达式就有一个在编译时已知的类型。确定表达式的类型的规则针对每种类型的表达式分别进行了阐释。

表达式的值是与表达式的类型兼容的赋值(5.2 节)，除非堆污染(4.12.2.1 节)发生。同样，存储在变量的值始终与变量的类型相兼容，除非堆污染发生。换句话说，类型为  $T$  的表达式始终适合于赋值给类型为  $T$  的变量。

注意，其类型是类类型  $F$  (声明为 `final`) 的表达式保证有一个值，该值要么是一个空引用，要么是一个其类是  $F$  本身的对象，因为 `final` 类型没有子类。

## 15.4 精确浮点数表达式

如果表达式的类型是 `float` 或 `double`，那么就有一个关于什么值集合(4.2.3 节)可以从中提取表达式的值的问题。这是由值集合转换(5.1.13 节)的规则控制的；这些规则依次取决于表达式是否是精确浮点数的(FP-strict)。

每个编译时常量表达式(15.28 节)是 FTP 严格的。如果表达式不是编译时常量表达式，那么就考虑包含表达式的所有类声明、接口声明和方法声明。如果任何这样的声明容忍 `strictfp` 修饰符，那么表达式就是精确浮点的。

如果一个类、接口或方法  $X$  被声明为 `strictfp`，那么  $X$  或  $X$  中的任何类、接口、方法和构造函数、实例初始化程序、静态初始化程序或变量初始化程序就是精确浮点数的。注意，一个注释(9.7 节)元素值(9.6 节)始终是精确浮点的，因为它始终是一个编译时常量(15.28 节)。

可以推出：当且仅当表达式不是一个编译时常量表达式，并且它没有出现在具有 `strictfp` 修饰符的任何声明中，那么表达式就不是精确浮点的。

在精确浮点数表达式中，所有中间值必须是 `float` 值集合或 `double` 值集合的元素，暗示所有精确浮点的表达式的结果必须是由 IEEE 754 算术在使用 `single` 和 `double` 格式表示的操作数上预测的结果。在不是精确浮点数的表达式中，一些余地保证了实现使用扩展的指数范围来表示中间的结果；大体而言，实际结果是：在 `float` 值集合或 `double` 值集合的排它使用可能导致上溢或下溢的情况下，计算可能产生“不正确的答案”。

## 15.5 表达式和运行时检查

如果表达式的类型是基本类型，那么表达式的值就是相同的基本类型。但如果表达式的类型是一个引用类型，那么被引用的对象的类，或者即使值是一个对象的引用而不是 `null`，在运行时是不需要知道的。在 Java 编程语言的少数几个地方，被引用的对象的实际类以不可以从表达式的类型中推导的方式影响程序执行。

- 方法调用 (15.12 节)。用于调用 `o.m(...)` 的特别方法是基于属于类型为 `o` 的类或接口的方法进行选择的。对于实例方法，`o` 的运行时值引用的对象的类参与了，因为子类可能重写已经在父类中声明的特定方法，以便调用该重写行为（重写方法可能或不可能选择进一步调用原始重写 `m` 方法）。
- `instanceof` 运算符 (15.20.2 节)。类型是引用类型的表达式可以使用 `instanceof` 进行测试，以查出表达式的运行时值引用的对象的类是否是某个其他引用类型相兼容的赋值。
- 强制转换 (5.5 节、15.16 节)。操作数表达式的运行时值引用的对象的类可能不与强制转换指定的类型相兼容。对于引用类型，这可能要求运行时检查，如果被引用的对象的类（如运行时确定的）不是与目标类型相兼容 (5.2 节) 的赋值，那么运行时检查就抛出异常。
- 赋值到引用类型的数组元素 (10.10 节、15.13 节、15.26.1 节)。如果 `S` 是 `T` 的子类型，类型检查规则允许将数组类型 `S[]` 当成 `T[]` 的子类型，但这要求运行时检查，以便赋值到数组元素，类似于针对强制转换执行的检查。
- 异常处理 (14.20 节)。异常是由 `catch` 子句捕获的，前提是被抛出的异常对象的类是 `catch` 子句的形式参数的类型的一个 `instanceof`。

对象的类不是静态已知的情形可能导致运行时类型错误。

此外，存在静态已知类型可能在运行时不是精确的情形。这样的一些情形可能在程序中产生未被检查的警告。给出这样的一些警告，以便响应不能静态保证是安全的，并且不能立即服从于动态检查的操作，因为它们涉及非泛型具体化 (4.7 节) 字节，结果，在程序执行过程中后面的动态检查可能检测到不一致性，并导致运行时类型错误。

运行时类型错误只可以在下面这些情形中发生：

- 在强制转换中，当操作数表达式的值引用的对象的实际类与强制转换运算符 (5.5 节、15.16 节) 指定的目标类型不兼容时；在这种情形下，抛出一个 `ClassCastException`。
- 隐式地引入编译器生成的转换，以确保在非泛型具体化类型上的操作的有效性。
- 在赋值给引用类型的数组元素中，当被赋予的值引用的对象的实际类与数据的实际运行类元素类型不兼容时 (10.10 节、15.13 节、15.26.1 节)；在这种情形中，抛出一个 `ArrayStoreException`。
- 当异常没有被任何 `catch` 处理程序 (11.3 节) 捕获时；在这种情形下，碰到异常的控制线程首先调用其线程组的方法 `uncaughtException`，然后终止。

## 15.6 计算的正常和突然结束

一切到此为止：突然到了尽头。

——William Wordsworth, 《Apology for the Foregoing Poems》(1831)

每个表达式有求值的正常模式，在这种模式中某些计算步骤被执行。下面几节描述了每种表达式求值的正常模式。如果所有的步骤在没有异常抛出的情况下执行，那么表达式则为正常结束。

但如果表达式的计算抛出一个异常，那么表达式则为突然结束。突然结束始终有一个相关的原因，即一个带指定值的 `throw`。

运行时异常是通过预定义的操作抛出的，如下所示：

- 如果内存不足的话，类实际创建表达式（15.9 节），数组创建表达式（15.10 节）或字符串串接运算符表达式（15.18.1 节）抛出一个 `OutOfMemoryError`。
- 如果任何维度表达式的值小于 0（15.10 节），那么数据创建表达式抛出 `NegativeArraySizeException`。
- 如果对象引用表达式的值是 `null`，那么字段访问（15.11 节）抛出 `NullPointerException`。
- 如果目标引用是 `null`，调用实例方法的方法调用表达式（15.12 节）抛出 `NullPointerException`。
- 如果数组引用表达式的值是 `null`，那么数组访问（15.13 节）抛出 `NullPointerException`。
- 如果数组索引表达式的值是负的或大于或等于数组的 `length`，那么数组访问（节 15.13）抛出一个 `ArrayIndexOutOfBoundsException`。
- 如果强制转换被发现在运行时是不允许的，那么强制转换（15.16 节）抛出一个 `ClassCastException`。
- 如果右边操作数表达式是 0，那么整数除（15.17.2 节）或整数余（15.17.3 节）运算符抛出 `ArithmeticException`。
- 赋值给引用类型（15.26.1 节）的引用类型的数组元素、方法调用（15.12 节）、前缀或后缀增量（15.14.2 节、15.15.1 节）或减量运算符（15.14.3 节、15.15.2 节）可能抛出 `OutOfMemoryError`，作为装箱转换的结果（5.1.7 节）。
- 如果被赋予的值与数组的元素类型不兼容的话，赋值到引用类型（15.26.1 节）的数组元素抛出一个 `ArrayStoreException`。

方法调用表达式也可能导致抛出异常，前提是一个异常发生了，导致方法体的执行突然结束。如果异常发生了，导致构造函数的执行突然结束，那么类实例创建表达式也可能导致抛出异常。在表达式的求值期间，各种链接和虚拟机错误也可能发生。根据它们的性质，这样的一些错误是难于预测和难于处理的。

如果异常发生了，那么在表达式的正常模式下求值的所有步骤完成前，一个或多个表达

式的求值可能被终止；这样的一些表达式则为突然结束。术语“正常结束”和“突然结束”也应用到语句的执行（14.1 节）。语句可能出于种种原因突然地结束，而不仅因为抛出异常。

如果表达式的求值要求对子表达式进行求值，那么子表达式的突然结束总是导致立即突然完成表达式本身，原因与父表达式是一样的，在正常模式中求值的所有后续步骤都不被执行。

## 15.7 求值顺序

凡事都要规规矩矩地按次序进行。

——《哥林多前书》第 14 章 40 节

Java 编程语言保证运算符的操作数在特定的求值顺序（也就是从左到右）中进行求值。

建议代码不要严重地依赖于该规范。当每个表达式至多包含一个副作用时，代码通常是更加清楚的，因为这是它的最外部操作，而且当代码不准确地依赖于哪个异常产生为表达式从左到右求值的结果时，代码也通常更加清楚。

### 15.7.1 首先对左边操作数求值

在右边操作数的任何部分被求值前，二元运算符的左边操作数是被完全求值的。例如，如果左边操作数包含对一个变量的赋值，并且右边的操作数包含对该变量的引用，那么引用产生的值将反映赋值首先发生的事实。

因此：

```
class Test {
    public static void main(String[] args) {
        int i = 2;
        int j = (i=3) * i;
        System.out.println(j);
    }
}
```

输出：

9

不允许它输出 6，而是输出 9。

如果运算符是一个复合赋值运算符（15.26.2 节），那么左边操作数的求值包括记住左边操作数表示的变量，并提取和保存那个变量的值，以便用在隐式组合操作中。因此，如下的测试程序：

```
class Test {
    public static void main(String[] args) {
        int a = 9;
        a += (a = 3); // first example
        System.out.println(a);
    }
}
```



```
int b = 9;
b = b + (b = 3); // second example
System.out.println(b);
}
}
```

输出:

```
12
12
```

因为在加的右边操作数被求值之前，两个赋值语句都提取和记住左边操作数的值，该值是 9，从而将变量设置到 3。不允许任一示例产生结果 6。注意，根据 ANSI/ISO 标准，对于 C 而言，这两个例子都有未定义行为。

如果二元运算符的左边操作数的求值突然结束，那么右边操作数就没有一部分已经被求值。

从而，测试程序：

```
class Test {
    public static void main(String[] args) {
        int j = 1;
        try {
            int i = forgetIt() / (j = 2);
        } catch (Exception e) {
            System.out.println(e);
            System.out.println("Now j = " + j);
        }
    }
    static int forgetIt() throws Exception {
        throw new Exception("I'm outta here!");
    }
}
```

输出:

```
java.lang.Exception: I'm outta here!
Now j=1
```

也就是，在右边的操作数被求值以及它将 2 内嵌赋值到 j 发生前，运算符/的左边操作数 forgetIt() 抛出一个异常。

### 15.7.2 在操作之前计算操作数

Java 编程语言也保证在操作数本身的任何部分被执行前，运算符的每个操作数（除了条件运算符&&、||和?:）看起来被完全求值。

如果二元运算符是一个整数除/（15.17.2 节）或整数余%（15.17.3 节），那么它的执行可能产生 ArithmeticException，但此异常只在二元运算符的两个操作数被求值后抛出，且要这些求值突然结束。



因此，例如程序：

```
class Test {
    public static void main(String[] args) {
        int divisor = 0;
        try {
            int i = 1 / (divisor * loseBig());
        } catch (Exception e) {
            System.out.println(e);
        }
    }
    static int loseBig() throws Exception {
        throw new Exception("Shuffle off to Buffalo!");
    }
}
```

始终输出：

```
java.lang.Exception: Shuffle off to Buffale!
```

而不输出：

```
java.lang.ArithmeticException: / by zero
```

因为在调用 `loseBig` 完成前，除操作没有一部分（包括发出被零除异常）可能看起来会发生，即使实现可能能够检测或推断除操作当然会导致被零除异常。

### 15.7.3 求值考虑括号和优先级

Java 编程语言实现必须考虑求值的顺序，该顺序由括号显式指出及由运算符优先级隐式指出。对于可能涉及的所有可能的计算值，实现可能不会利用代数标识（比如结合律）来将表达式重写到一个更加方便的计算顺序，除非它可以证明替换表达式在值及在它的可观察副作用中是等价的，甚至存在执行的多个线程（在第 17 章中使用线程执行模型）。

在浮点计算的情形中，此规则也应用于无穷大和非数字（NaN）值。例如， $!(x < y)$  可能不会被重写为  $x \geq y$ ，因为如果  $x$  或  $y$  是 NaN 或二者都是 NaN，这些表达式就有不同的值。

特别地，看起来算术结合的浮点计算不像是计算结合的。这样的一些计算必须不是单纯地进行重排序。

例如，让 Java 编译器重写  $4.0 * x * 0.5$  作为  $2.0 * x$  是不正确的：尽管舍入在这里碰巧不是一个问题，但有  $x$  的大值，对于这些值，第一个表达式产生无穷大（因为上溢），但第二个表达式产生有限结果。

因此，例如下面的测试程序：

```
strictfp class Test {
    public static void main(String[] args) {
        double d = 8e+307;
        System.out.println(4.0 * d * 0.5);
    }
}
```

```
        System.out.println(2.0 * d);
    }
}
```

输出:

```
Infinity
1.6e+308
```

因为第一个表达式上溢, 第二个没有溢出。

比较而言, 整数加和乘在 Java 编程语言中是可证明结合的。

例如,  $a+b+c$ , 其中  $a$ 、 $b$  和  $c$  是局部变量 (这种简化假定避免了涉及多线程和 `volatile` 变量的问题), 将始终产生相同的答案, 而不管是作为  $(a+b)+c$  或  $a+(b+c)$  求值: 如果表达式  $b+c$  在代码中的附近出现, 那么智能的编译器就可能能够使用这种共同的子表达式。

#### 15.7.4 参数清单从左到右进行计算

在方法或构造函数调用或类实例创建表达式中, 参数表达式可能出现在括号中, 并由逗号分开。每个参数表达式在它的左边的任何参数表达式的任何部分之前表现为完全计算。因此:

```
class Test {
    public static void main(String[] args) {
        String s = "going, ";
        print3(s, s, s = "gone");
    }
    static void print3(String a, String b, String c) {
        System.out.println(a + b + c);
    }
}
```

始终输出:

```
going, going, gone
```

因为将字符串 "gone" 赋值给  $s$  是在已经计算 `print3` 的最初两个参数之后发生的。

如果参数表达式的求值突然结束, 那么它的右边就没有任何参数表达式的任何部分表现为已经计算。

从而, 例子:

```
class Test {
    static int id;
    public static void main(String[] args) {
        try {
            test(id = 1, oops(), id = 3);
        } catch (Exception e) {
            System.out.println(e + ", id=" + id);
        }
    }
}
```

```
static int oops() throws Exception {  
    throw new Exception("oops");  
static int test(int a, int b, int c) {  
    return a + b + c;  
}  
}
```

输出:

Java.lang.Exception:oops, id=1

因为将 3 赋值给 id 不执行。

### 15.7.5 其他表达式的求值顺序

一些表达式的求值顺序并没有被这些泛型规则所完全覆盖,因为这些表达式可能发生异常的条件数次,这些次数必须被指定。请特别看一下下面几种表达式的求值的详细解释:

- 类实例创建表达式 (15.9.4 节)
- 数组创建表达式 (15.10.1 节)
- 方法调用表达式 (15.12.4 节)
- 数组访问表达式 (15.13.1 节)
- 涉及数组元素的赋值 (15.26 节)

## 15.8 主表达式

主表达式包括大多数最简单类别的表达式,从这些表达式中可构造所有其他的表达式:文本、类文本、字段访问、方法调用和数组访问。带括号的表达式也从句法上被当成主表达式。

*primary:*

*PrimaryNoNewArray*

*ArrayCreationExpression*

*PrimaryNoNewArray:*

*Literal*

*Type . class*

*void . class*

*this*

*ClassName.this*

*( Expression )*

*ClassInstanceCreationExpression*

*FieldAccess*

*MethodInvocation*

*ArrayAccess*

### 15.8.1 词法字面值

字面值 (3.10 节) 表示一个固定的、没有改变的值。

下面所示为从第 3.10 节中产生的, 在这里为方便起见进行了重复:

*Literal:*

*IntegerLiteral*

*FloatingPointLiteral*

*BooleanLiteral*

*CharacterLiteral*

*StringLiteral*

*NullLiteral*

字面值的类型按如下所示进行确定:

- 以 L 或 l 结束的整数是 long; 任何其他整数的类型是 int。
- 以 F 或 f 结束的浮点数的类型是 float, 并且它的值必须是 float 值集合 (4.2.3 节) 的一个元素。任何其他浮点数的类型是 double, 并且它的值必须是 double 值集合的一个元素。
- 布尔值的类型是 boolean。
- 字符的类型是 char。
- 字符串的类型是 String。空值 null 的类型是 null 类型; 其值是一个空引用。

词法字面值的计算始终是正常结束的。

### 15.8.2 class 字面值

class 字面值是一个表达式, 该表达式包括类的名字、接口、数组或基本类型或伪类型 void, 跟上一个 “.” 及符号 class。class 字面值的类型 C.Class (其中 C 是类、接口或数组类型的名称) 是 Class<C>。如果 p 是基本类型的名称, 令 B 是装箱转换 (5.1.7 节) 后类型为 p 的表达式类型。void.class 的类型是 Class<void>。

class 常量求值等于指定类型 (或 void) 的 Class 对象, 该类型通过定义当前实例的类的类加载器定义的。

如果下面的任何一种情况发生, 那它就是一个编译时错误:

- 指定的类型是一个类型变量 (4.4 节) 或参数化类型 (4.5 节) 或者一个数组, 其中元素类型是一个类型变量或参数化类型。
- 指定的类型不表示类型是访问的 (6.6 节), 且类型在 class 常量出现的点的作用域 (6.3 节) 中。

### 15.8.3 this

关键字 this 只可以用在实例方法、实例初始化程序或构造函数的体中, 或在类的实例变量的初始化程序中。如果它在其他任何地方出现, 就会出现一个编译时错误。

当用作主表达式时，关键字 `this` 表示是调用实例方法的对象引用的值，或者表示是正被构造的对象的引用的值。`this` 的类型是类 `C`，在该类中关键字 `this` 出现了。在运行时，被引用的实际对象的类是类 `C` 或 `C` 的任何子类。

在下面的示例中：

```
class IntVector {
    int[] v;
    boolean equals(IntVector other) {
        if (this == other)
            return true;
        if (v.length != other.v.length)
            return false;
        for (int i = 0; i < v.length; i++)
            if (v[i] != other.v[i])
                return false;
        return true;
    }
}
```

类 `IntVector` 实现了方法 `equals`，该方法比较两个向量。如果 `other` 向量是与 `equals` 方法被调用的向量对象相同的向量对象，那么检查可以跳过长度和值比较。`equals` 方法通过将对 `other` 对象的引用与 `this` 比较来实现此检查。

关键字 `this` 也用在特殊显式构造函数调用语句中，该语句可以出现在构造函数体的开始处（8.8.7 节）。

#### 15.8.4 限定的 `this`

任何词法上关闭的实例可以通过显式限制关键字 `this` 进行引用。

令 `C` 是 `ClassName` 表示的一个类。令 `n` 是一个整数，使得 `C` 是类的第 `n` 个词法上封闭的类，在该类中，限制的 `this` 表达式出现了。形如 `ClassName.this` 的表达式值是 `this` 的第 `n` 个词法上关闭的实例（8.1.3 节）。表达式的类型是 `C`。如果当前类不是类 `C` 的一个内部类或 `C` 本身，那么它就是一个编译时错误。

#### 15.8.5 带括号的表达式

带括号表达式是一个主表达式，其类型是被包含的表达式类型，且其在运行时的值是被包含表达式的值。如果被包含的表达式表示一个变量，那么带括号表达式也表示该变量。

使用括号只影响求值的顺序，且有一个迷人的异常。



考虑如果是下面情形的话看类型 `long` 是否有最小可能负值。此值 `9223372036854775808L` 只被允许作为一元减运算符（3.10.1 节）的一个操作数。因此，在括号中封闭它，如 `-(9223372036854775808L)`，将导致一个编译时错误。

在表达式周围存在或缺少括号尤其不会（除了上面提及的情形）影响：

- 类型为 `float` 或 `double` 的表达式值的值集合（4.2.3 节）的选择。
- 变量是否被确定赋值、是否当真时确定赋值、当假时确定赋值、确定不赋值、当真时确定不赋值或当假时确定不赋值（第 16 章）。

## 15.9 类实例创建表达式

现在有一个新的目标占据了我的心头。

——艾德嘉·爱伦·坡，《A Tale of the Ragged Mountains》（1844）

类实例创建表达式用于创建是类的实例的新对象。

*ClassInstanceCreationExpression*:

`new TypeArgumentsopt ClassOrInterfaceType ( ArgumentListopt)`

*ClassBody<sub>opt</sub>*

`Primary. new TypeArgumentsopt Identifier TypeArgumentsopt (`

`ArgumentListopt) ClassBodyopt`

*ArgumentList*:

`Expression`

`ArgumentList , Expression`

类实例创建表达式指定要被实例化的类，可能跟有类型参数[如果正被初始化的类是泛型的（8.1.2 节）]，跟有（一个可能为空的）实际值参数的列表给构造函数。也可能将显式类型参数传递给构造函数[如果它是泛型构造函数（8.8.4 节）]。构造函数的类型参数立即跟上关键字 `new`。如果类实例创建表达式中使用的任何类型参数是通配符类型参数（4.5.1 节），它就是一个编译错误。类实例创建表达式具有两种形式：

- 非限定类实例创建表达式以关键字 `new` 开始。非限定类实例创建表达式可用于创建类的一个实例，而不管类是否是顶层的（7.6 节）、成员（8.5 节、9.5 节）、本地的（14.3 节）或匿名的类（15.9.5 节）。
- 限定类实例创建表达式以 *Primary* 开始。限定类实例创建表达式使创建内部成员类及其匿名子类成为可能。

当且仅当为下面的情况之一时，类实例创建表达式可能抛出一个类型为 *E* 的异常：

- 表达式是一个限定类实例创建表达式，并且限定表达式可以抛出异常 *E*；或者
- 参数清单的某个表达式可以抛出 *E*；或者
- *E* 在被调用的构造函数的类型的抛出子句中列出；或者
- 类实例创建表达式包含一个 *ClassBody*，且 *ClassBody* 中一些实例初始化程序块或实例变量初始化表达式可能抛出 *E*。

非限定和限定的类实例创建表达式两者都可以选择性地用一个类体结束。这样的—



类实例创建表达式声明一个匿名类（15.9.5 节），并创建了它的一个实例。

我们说，当类的一个实例由类实例创建表达式创建时，类就被实例化了。类实例化包括确定什么类要被实例化，新创建实例的封闭实例（如果有的话）是什么，什么构造函数应该被调用以创建新的实例，以及哪些参数应该被传递给该构造函数。

### 15.9.1 确定正被实例化的类

如果类实例创建表达式在类体中结束，则正被实例化的类就是一个匿名的类。那么：

- 如果类实例创建表达式是一个非限定类实例创建表达式，那么令  $T$  是 *new* 标记之后的 *ClassOrInterface*。如果  $T$  命名的类或接口是不可访问的（6.6 节），或者如果  $T$  是一个枚举类型（8.9 节）那么它就是一个编译时错误。如果  $T$  表示一个类，那么  $T$  命名的类的匿名直接子类就会被声明。如果由  $T$  表示的类是一个 *final* 类，那么它就是一个编译时错误。如果  $T$  表示一个接口，那就声明了实现  $T$  命名的接口的 *Object* 的匿名直接子类。在任何一种情形中，子类的体就是类实现创建表达式中给出的 *ClassBody*。正被实例化的类是匿名子类。
- 否则，类实例创建表达式是一个限定的类实例创建表达式。令  $T$  是 *new* 标记之后的标识符的名称。如果  $T$  不是一个可访问（6.6 节）非最终内部类（8.13 节）（该类是 *Primary* 编译时类型的一个成员）的简单名称（6.2 节），那么它就是一个编译时错误。如果  $T$  是不明确（8.5 节）或  $T$  表示一个枚举类型，那么它也是一个编译时错误。声明了一个  $T$  命名的类的匿名直接子类。子类的体是在类实例创建表达式中给出的 *ClassBody*。正被初始化的类是一个匿名子类。

如果类实例创建表达式没有声明匿名类，那么：

- 如果类实例创建表达式是一个非限定类实例创建表达式，那么 *ClassOrInterfaceType* 必须表示一个可访问的类（6.6 节），并且该类不是枚举类型，也不是 *abstract*，否则出现编译时错误。在这种情形中，正被初始化的类是由 *ClassOrInterfaceType* 表示的类。
- 否则，类实例创建表达式是一个限定的类实例创建表达式。如果 *Identifier* 不是可访问（6.6 节）非 *abstract* 内部类  $T$ （8.1.3 节）的简单名称（6.2 节），那么它是一个编译时错误，类  $T$  是 *Primary* 的编译时类型的一个成员。如果 *Identifier* 是不确定的（8.5 节），或者如果 *Identifier* 表示枚举类型（8.9 节），那么它也是一个编译时错误。正被实例化的类是 *Identifier* 表示的类。

类实例创建表达式的类型是正被实例化的类类型。

### 15.9.2 确定封闭实例

令  $C$  是正被实例化的类，令  $i$  是正被创建的实例。如果  $C$  是一个内部类，那么  $i$  可能有一个立即封闭的实例。 $i$  的立即封闭的实例（8.1.3 节）按如下进行确定：

- 如果  $C$  是一个匿名类，那么：
  - ◆ 若类实例创建表达式在静态的上下文中出现（8.1.3 节），那么  $i$  没有立即封闭的实例。

- ◆ 否则,  $i$  的立即封闭实例是 `this`。
- 如果  $C$  是一个本地类 (1.4.3 节), 那么令  $O$  是  $C$  的最内部词法上封闭的类。令  $n$  是一个整数, 使得  $O$  是类的第  $n$  个词法上封闭的类, 在该类中出现了类实例创建表达式。那么:
  - ◆ 若  $C$  在静态上下文中出现, 那么  $i$  没有立即封闭的实例。
  - ◆ 否则, 如果类实例创建表达式在表达式上下文中出现, 那么就发生编译时错误。
  - ◆ 否则,  $i$  的立即封闭实例是 `this` 的第  $n$  个词法上封闭的实例 (8.1.3 节)。
- 否则  $C$  是一个内部成员类 (8.5 节)。
  - ◆ 若类实例创建表达式是一个非限定类实例创建表达式, 那么:
    - ◇ 如果类实例创建表达式在静态上下文中出现, 那么就发生编译时错误。
    - ◇ 否则, 如果  $C$  是封闭类一个成员, 那么令  $O$  是词法上最内部的封闭类,  $C$  是这个类的一个成员, 并令  $n$  是一个整数, 使得  $O$  是类的第  $n$  个词法上封闭的类, 类中出现了类实例创建表达式。 $i$  的立即封闭的实例是 `this` 的第  $n$  个词法上封闭的实例。
    - ◇ 否则, 一个编译时错误发生。
  - ◆ 否则, 类实例创建表达式是一个限定的类实例创建表达式。 $i$  的立即封闭实例是这样的一个对象, 该对象是 *Primary* 表达式的值。

此外, 如果  $C$  是一个匿名类, 并且  $C$  的直接超类  $S$  是一个内部类, 那么  $i$  可能有一个关于  $S$  的直接封闭类, 该类定义如下:

- 如果  $S$  是一个本地类 (14.3 节), 那么令  $O$  是  $S$  的最内部词法上封闭的类。令  $n$  是一个整数, 使得  $O$  是类实例创建表达式出现所在的类的第  $n$  个词法上封闭的类。
  - ◆ 若  $S$  在静态上下文中出现, 那么  $i$  就没有与  $S$  有关的立即封闭实例。
  - ◆ 否则, 如果类实例创建表达式在静态上下文中出现, 那么就会发生编译时错误。
  - ◆ 否则, 与  $S$  有关  $i$  的立即封闭实例就是 `this` 的第  $n$  个词法上封闭的实例。
- 否则,  $S$  是一个内部成员类 (8.5 节)。
  - ◆ 若类实例创建表达式是一个未限定的类实例创建表达式, 那么:
    - ◇ 如果类实例创建表达式在静态上下文中出现, 则发生一个编译时错误。
    - ◇ 否则, 如果  $S$  是封闭类的一个成员, 那么令  $O$  是一个最内部的词法上封闭的类,  $S$  是这个类的一个成员, 并令  $n$  是一个整数, 使得  $O$  是其中出现类实例创建表达式的类的第  $n$  个词法上封闭的类。与  $S$  有关的  $i$  的立即封闭实例是 `this` 的第  $n$  个词法上封闭的实例。
    - ◇ 否则, 发生一个编译时错误。
  - ◆ 否则, 类实例创建表达式是一个限定的类实例创建表达式。与  $S$  有关的  $i$  的立即封闭实例是 *Primary* 表达式的值的对象。

### 15.9.3 选择构造函数及其参数

令  $C$  是正被实例化的类类型。要创建  $C$  的一个实例, 通过下面的规则在编译时选择  $C$

的构造函数：

- 首先，确定构造函数调用的实例参数。
  - ◆ 如果  $C$  是一个匿名类，并且  $C$  的直接超类  $S$  是一个内部类，那么：
    - ◇ 若  $S$  是一个本地类，并且  $S$  在静态上下文中出现，那么参数清单中的参数（如果有的话）就是构造函数的参数（按照它们在表达式中出现的顺序）。
    - ◇ 否则，与  $S$  有关的  $i$  的立即封闭实例是构造函数的第一个参数，跟上类实例创建表达式的参数清单中的参数（如果有的话），并按照它们在表达式中出现的顺序。
  - ◆ 否则，参数清单中的参数（如果有的话）是构建函数的参数，并按照它们在表达式中出现的顺序。
- 一旦确定实际参数，它们就用于选择  $C$  的构造函数，使用与用于方法调用相同的规则（15.12 节）。如方法调用中一样，编译时方法匹配错误导致是否没有既可应用又可访问的惟一最特定的构造函数。

注意，类实例创建表达式的类型可能是匿名类类型，在这种情形中，被调用的构造函数是一个匿名构造函数。

#### 15.9.4 类实例创建表达式的运行时计算

在运行时，类实例创建表达式的计算如下所示。

首先，如果类实例创建表达式是一个限定的类实例创建表达式，那就计算限定主表达式。如果限定表达式求值为 `null`，就引发一个 `NullPointerException`，并且类实例创建表达式突然结束。如果限定表达式突然结束，类实例创建表达式就出于相同的原因突然结束。

接下来，分配空间给新的类实例。如果没有足够的空间分配给对象，类实例创建表达式的求值就通过抛出一个 `OutOfMemoryError` 突然结束（15.9.6 节）。

新对象包含指定类类型中声明的所有字段的新实例及所有它的超类。当创建每个新的字段实例时，它被初始化到它的默认值（4.12.5 节）。

接下来，对构造函数的实际参数进行从左到右求值。如果任何参数求值突然结束，那么它右边的任何参数表达式就不被求值，并且出于相同的原因，类实例创建表达式突然结束。

接下来，调用指定类类型的选定构造函数。这导致了至少对类类型的每个超类调用一个构造函数。这个进程可能通过显式构造函数调用语句进行引导（8.8 节），并且在第 12.5 节中会详细进行描述。

类实例创建表达式的值是指定类的新创建对象的一个引用。每次对表达式进行求值，会创建一个新的对象。

#### 15.9.5 匿名类声明

匿名类声明自动通过编译器从类实例创建表达式派生出来。

匿名类从来都不是 `abstract` (8.1.1.1 节)。匿名类始终是一个内部类 (8.1.3 节)；它从未是 `static` (8.1.1 节, 8.5.2 节)。匿名类始终是隐式的 `final` (8.1.1.2 节)。

#### 15.9.5.1 匿名构造函数

匿名类不可以有显式声明的构造函数类。相反，编译器必须自动提供匿名类的匿名构造函数。具有直接超类 *S* 的匿名类 *C* 的匿名构造函数的形式如下所示：

- 如果 *S* 不是一个内部类，或如果 *S* 是一个在静态上下文中出现的本地类，那么匿名类构造函数对于每个声明 *C* 的类实例创建表达式的实际参数有一个形参。类实例创建表达式的实参用于确定 *S* 的构造函数 *cs*，所使用的规则与用于方法调用的规则相同 (15.12 节)。匿名构造函数的每个形参的类型必须等于 *cs* 的相应形参。

构造函数体包括一个形如 `super(...)` 的显式构造函数调用 (8.8.7.1 节)，其中实参是构造函数的形参，并且按照它们被声明的顺序。

- 否则，*C* 的构造函数的第一个形参表示与 *S* 有关的 *i* 的立即封闭的实例的值。这种参数的类型是立即封闭 *S* 的声明的类类型。对于声明匿名类的类实例创建表达式的每个实参，构造函数有另外一个形参。第 *n* 个形参 *e* 对应于 *n-1* 个实参。类实例创建表达式的实参用于确定 *S* 的构造函数 *cs*，使用与方法调用相同的规则 (15.12 节)。匿名构造函数的每个形参的类型必须等于 *cs* 的相应形参。构造函数体包含形如 `o.super(...)` 的显式构造函数调用 (8.8.7.1 节)，其中 *o* 是构造函数的第一个形参，并且实参是构造函数的后续形参，按照它们被声明的顺序。

在所有的情形中，匿名构造函数的 `throws` 子句必须列出包含在匿名构造函数中的显式超类构造函数调用语句抛出的所有检查异常，以及抛出匿名类的任何实例初始化程序或实例变量初始化抛出的所有检查异常。

注意，可以让匿名构造函数的签名引用一个不可访问的类型（例如，如果这样的一种类型在超类构造函数的签名 *cs* 中出现）。在编译时或运行时，这本身不导致任何错误。

#### 15.9.6 示例：求值顺序和内存不足检测

如果一个类实例创建表达式的求值查出没有足够的内存来执行创建操作，那么就抛出一个 `OutOfMemory`。在对任何参数表达式求值前，这种检查就发生了。

因此，例如下面的测试程序：

```
class List {
    int value;
    List next;
    static List head = new List(0);
    List(int n) { value = n; next = head; head = this; }
}
class Test {
    public static void main(String[] args) {
        int id = 0, oldid = 0;
        try {
            for (;;) {
```

```

        ++id;
        new List(olddid = id);
    }
} catch (Error e) {
    System.out.println(e + ", " + (olddid==id));
}
}
}

```

输出:

```
java.lang.OutOfMemoryError: List, false
```

因为在参数表达式 `olddid=id` 被求值前, 内存不足条件被检测到了。

将它与数组创建表达式的处理比较 (15.10 节), 在对维度表达式 (15.10.3 节) 求值后, 这些表达式的内存不足条件就被检测到。

## 15.10 数组创建表达式

该有的都有了, 我编织着新的梦想……  
——查尔斯·狄更斯, 《远大前程》(1861)

数组实例数组表达式用于创建新的数组 (第 10 章)。

*ArrayCreationExpression*:

```

new PrimitiveType DimExprs Dimsopt
new ClassOrInterfaceType DimExprs Dimsopt
new PrimitiveType Dims ArrayInitializer
new ClassOrInterfaceType Dims ArrayInitializer

```

*DimExprs*:

```

DimExpr
DimExprs DimExpr

```

*DimExpr*:

```
[Expression]
```

*Dims*:

```

[]
Dims []

```

数组创建表达式创建一个对象, 该对象是一个新的数组, 其元素的类型是 *PrimitiveType* 或 *ClassOrInterfaceType* 指定的类型。如果 *ClassOrInterfaceType* 没有表示泛型具体化类型 (4.7 节), 那么它就是一个编译时错误。否则, *ClassOrInterfaceType* 可能命名任何指定的引用类型, 甚至 *abstract* 类类型 (8.1.1.1 节) 或接口类型 (第 9 章)。



上面的规则暗示着数组创建表达式中的元素类型不可能是一个参数化类型，除了一个没有限制的通配符。

创建表达式的类型是一个数组类型，这种类型不能通过创建表达式的副本进行表示，`new` 关键字和每个 `DimExpr` 表达式及数组初始化程序已经从该表达式中被删除。

例如，创建表达式的类型：

```
new double[3][3][ ]
```

是：

```
double[][][]
```

`DimExpr` 中的每个维度表达式的类型必须是一个可以转换（5.1.8 节）到整型的类型，否则会发生编译时错误。每个表达式均经历一元数值提升。被提升的类型必须是一个 `int`，否则会发生编译时错误；这特别意味着，维度表达式的类型必须不为 `long`。

如果提供了数组初始化程序，那么新分配的数组将用数组初始化程序提供的值进行初始化，如第 10.6 节中所描述。

### 15.10.1 数据创建表达式的运行时计算

在运行时，数组表达式的计算表现如下。如果没有维度表达式，那么就必须有一个数组初始化程序。数组初始化程序的值是数组创建表达式的值。否则：

首先，维度表达式从左到右进行求值。如果任何表达式突然结束，那么它右边的表达式就不进行求值。

接下来，维度表达式的值进行检查。如果 `DimExpr` 表达式的值小于 0，那么就抛出 `NegativeArraySizeException`。

接下来，为新数组分配空间。如果没有足够的空间分配给数组，数组创建表达式的求值就通过抛出 `OutOfMemoryError` 突然结束。

然后，如果一个单独的 `DimExpr` 出现，那么一维数组用指定的长度创建，并且数组的每个元素被初始化到默认值（4.12.5 节）。

如果数组创建表达式包含  $N$  个 `DimExpr` 表达式，那么它就有效地执行深度为  $N-1$  的一组嵌套循环，以便创建数组的隐含数组。

例如，声明：

```
float[] matrix = new float[3][3]
```

在行为上等于：

```
float[][] matrix = new float[3][ ];  
for (int d = 0; d < matrix.length; d++)  
    matrix[d] = new float[3];
```

和：



```
Age[][][][][] Aquarius = new Age[6][10][8][12][];
```

等价于：

```
Age[][][][][] Aquarius = new Age[6][][][][];
for (int d1 = 0; d1 < Aquarius.length; d1++) {
    Aquarius[d1] = new Age[10][][][];
    for (int d2 = 0; d2 < Aquarius[d1].length; d2++) {
        Aquarius[d1][d2] = new Age[8][][];
        for (int d3 = 0; d3 < Aquarius[d1][d2].length; d3++) {
            Aquarius[d1][d2][d3] = new Age[12][];
        }
    }
}
```

且 d、d1、d2 和 d3 由已经不在本地声明的名称替换了。从而，单独一个表达式实际上已经创建了长度为 6 的一个数组，6 个长度为 10 的数组， $6 \times 10 = 60$  个长度为 8 的数组，及  $6 \times 10 \times 8 = 480$  个长度为 12 的数组。此例子中留下了第 5 维，该维度会是包含实际数组元素的数组（引用 Age 对象），这些元素只被初始化到空引用。这些数组可以通过其他地址在后面进行填充，比如：

```
Age[] Hair = { new Age("quartz"), new Age("topaz") };
Aquarius[1][9][6][9] = Hair;
```

多维数组不需要在每一层上有相同长度的数组。

因此，三角矩阵可以通过下面的代码创建：

```
float triang[][] = new float[100][];
for (int i = 0; i < triang.length; i++)
    triang[i] = new float[i+1];
```

### 15.10.2 示例：数据创建求值顺序

在数组创建表达式（15.10 节）中，可能有一个或多个维度表达式，每个维度表达式在一些括号中。每个维度表达式完全在它右边的任何维度表达式的任何部分之前进行求值。从而：

```
class Test {
    public static void main(String[] args) {
        int i = 4;
        int ia[][] = new int[i][i-3];
        System.out.println(
            "[" + ia.length + ", " + ia[0].length + "]");
    }
}
```

输出：

```
[4, 3]
```

因为在第二个维度表达式将 *i* 设置成 3 之前，第一个维度被计算为 4。

如果维度表达式的求值突然结束，它右边的任何维度表达式的任何部分都将不表现为已经进行求值。因此，示例：

```
class Test {
    public static void main(String[] args) {
        int[][] a = { { 00, 01 }, { 10, 11 } };
        int i = 99;
        try {
            a[val()][i = 1]++;
        } catch (Exception e) {
            System.out.println(e + ", i=" + i);
        }
    }
    static int val() throws Exception {
        throw new Exception("unimplemented");
    }
}
```

输出：

```
java.lang.Exception: unimplemented, i=99
```

因为将 *i* 设置到 1 的内嵌赋值从未被执行。

### 15.10.3 示例：数组创建和内存不足检测

如果任何数组创建表达式的求值发现没有足够的内存来执行创建执行，那么就抛出 `OutOfMemoryError`。只有在所有维度表达式的求值已经正常结束后，这种检查才发生。因此，例如下面的测试程序：

```
class Test {
    public static void main(String[] args) {
        int len = 0, oldlen = 0;
        Object[] a = new Object[0];
        try {
            for (;;) {
                ++len;
                Object[] temp = new Object[oldlen = len];
                temp[0] = a;
                a = temp;
            }
        } catch (Error e) {
            System.out.println(e + ", " + (oldlen==len));
        }
    }
}
```

输出：

```
java.lang.OutOfMemoryError, true
```

因为在维度表达式 `oldlen=len` 被求值后会检测到内存不足的条件。

将它与类实例创建表达式（15.9 节）比较，在对参数表达式（15.9.6 节）进行求值前，类实例创建表达式会检测到内存不足的条件。

## 15.11 字段访问表达式

字段访问表达式可以访问对象或数组的一个字段——表达式或特殊关键字 `super` 的值的引用（它也可以通过使用简单的名称引用当前实例或当前类的字段；参见第 6.5.6 节）。

*FieldAccess:*

*Primary* . *Identifier*

`super` . *Identifier*

*ClassName* . `super` . *Identifier*

字段访问表达式的含义通过将相同的规则用于限定名称（6.6 节）来确定的，但由下面的事实确定：表达式不能表示一个包、类类型或接口类型。

### 15.11.1 使用 *Primary* 的字段访问

*Primary* 的类型必须是一个引用类型 *T*，否则，发生一个编译时错误。字段访问表达式的含义如下进行定义：

- 如果标识符命名了类型为 *T* 的几个可访问成员字段，那么字段访问就是不明确的，并发生编译时错误。
- 如果标识符没有命名一个类型为 *T* 的可访问成员字段，那么字段访问是未定义的，并且发生编译时错误。
- 否则，标识符命名类型为 *T* 的单个可访问成员字段，并且字段访问表达式的类型是捕获转换（5.1.10 节）后成员字段的类型。在运行时，字段访问表达式的结果进行如下计算：
  - ◆ 若字段是 `static`：
    - ◇ *Primary* 表达式被求值，并且结果被丢弃。如果 *Primary* 表达式的求值突然结束，那么字段访问表达式会出于相同的原因突然结束。
    - ◇ 如果字段是 `final`，那么结果是类型为 *Primary* 表达式的类或接口中指定的类变量的值。
    - ◇ 如果字段不是 `final`，那么结果是一个变量，也就是类型为 *Primary* 表达式的类中的指定类变量。
  - ◆ 若字段不是 `static`：
    - ◇ *Primary* 表达式被求值。如果 *Primary* 表达式的求值被突然结束，那么字段访问表达式会出于相同的原因突然结束。
    - ◇ *Primary* 的值是 `null`，那么就抛出 `NullPointerException`。

- ✧ 如果字段是 `final`，那么结果就是 *Primary* 的值引用的对象中指定的实例变量。
- ✧ 如果字段不是 `final`，那么结果就是一个变量，也就是，*Primary* 的值引用的对象中指定的实例变量。

请特别注意，只有 *Primary* 表达式的类型，而不是运行时引用的实际对象的类用于确定要使用哪个字段。

从而，示例：

```
class S { int x = 0; }
class T extends S { int x = 1; }
class Test {
    public static void main(String[] args) {
        T t = new T();
        System.out.println("t.x=" + t.x + when("t", t));
        S s = new S();
        System.out.println("s.x=" + s.x + when("s", s));
        s = t;
        System.out.println("s.x=" + s.x + when("s", s));
    }

    static String when(String name, Object t) {
        return "when " + name + " holds a "
            + t.getClass() + " at run time.";
    }
}
```

产生如下输出：

```
t.x=1 when t holds a class T at run time.
s.x=0 when s holds a class S at run time.
s.x=1 when s holds a class T at run time.
```

最后一行展示了被访问的字段实际上没有依赖于被引用的对象的运行时类；即使 `s` 持有对类 `T` 的对象的一个引用，表达式 `s.x` 也引用类 `S` 的 `x` 字段，因为表达式 `s` 的类型是 `S`。类 `T` 的对象包含两个名为 `x` 的字段，一个针对类 `T`，一个针对它的超类 `S`。

缺少字段访问的动态查询允许程序用简单的实现来高效地运行。后期绑定和重写的功能是可用的，但只有在使用实例方法时可用。考虑相同的示例，该示例使用实例方法来访问字段：

```
class S { int x = 0; int z() { return x; } }
class T extends S { int x = 1; int z() { return x; } }
class Test {
    public static void main(String[] args) {
        T t = new T();
        System.out.println("t.z()=" + t.z() + when("t", t));
        S s = new S();
        System.out.println("s.z()=" + s.z() + when("s", s));
    }
}
```

```

    s = t;
    System.out.println("s.z()=" + s.z() + when("s", s));
}
static String when(String name, Object t) {
    return "when " + name + " holds a "
        + t.getClass() + " at run time.";
}
}

```

现在输出是：

```

t.z()=1 when t holds a class T at run time.
s.z()=0 when s holds a class S at run time.
s.z()=1 when s holds a class T at run time.

```

最后一行展示了被访问的方法实际上没有依赖于被引用对象的运行时类：当 *s* 持有类 *T* 的对象的一个引用时，表达式 *s.z()* 引用类 *T* 的 *z* 方法，而不管表达式 *s* 的类型是 *S* 的事实。类 *T* 的方法 *z* 重写了类 *S* 的方法 *z*。

下面的示例展示了一个空引用可能用于访问一个类（*static*）变量，而不会发生异常：

```

class Test {
    static String mountain = "Chocorua";
    static Test favorite(){
        System.out.print("Mount ");
        return null;
    }
    public static void main(String[] args) {
        System.out.println(favorite().mountain);
    }
}

```

它编译、执行并输出：

```
Mount Chocorua
```

即使 *favorite()* 的结果是 *null*，也不会抛出 *NullPointerException*。“Mount”被输出展示 *Primary* 表达式在运行时确实完全地进行求值，而不管下面的事实：只有它的类型，而不是它的值用于确定要访问哪个字段（因为字段 *mountain* 是 *static*）。

### 15.11.2 使用 *super* 访问超类成员

使用关键字 *super* 的特殊形式只有在实例方法、实例初始化程序或构造函数，或者在类的实例变量的初始化程序中是有效的；正好有可以使用关键字 *this* 的一些情形（15.8.3 节）。涉及 *super* 的形式不可以用在类 *Object* 中的任何地方，因为 *Object* 没有超类：如果 *super* 出现在类 *Object* 中，那么就导致一个编译时错误。

假设字段访问表达式 *super.name* 出现在类 *C* 中，并且 *C* 的直接超类是类 *S*。那么 *super.name* 正被当成是表达式 *((S) this).name*；因此，它引用当前对象的名为 *name* 字段，但当前的对象被视为超类的一个实例。从而它可以访问在类 *S* 中可见的名为 *name*

字段，即使那个字段通过在类 *C* 中声明名为 *name* 的字段来隐藏该字段。

*super* 的使用通过下面的例子来展示：

```
interface I { int x = 0; }
class T1 implements I { int x = 1; }
class T2 extends T1 { int x = 2; }
class T3 extends T2 {
    int x = 3;
    void test() {
        System.out.println("x=\t\t"+x);
        System.out.println("super.x=\t\t"+super.x);
        System.out.println("((T2)this).x=\t"+((T2)this).x);
        System.out.println("((T1)this).x=\t"+((T1)this).x);
        System.out.println("((I)this).x=\t"+((I)this).x);
    }
}
class Test {
    public static void main(String[] args) {
        new T3().test();
    }
}
```

上面代码产生下面的输出：

```
x=          3
super.x=    2
((T2)this).x=2
((T1)this).x=1
((I)this).x= 0
```

在类 *T3* 中，表达式 *super.x* 恰好被当成下面那样：

```
((T2)this).x
```

假设 *T.super.name* 的字段访问出现在类 *C* 中，并且由 *T* 表示的类的直接超类是完全限定名称为 *S* 的一个类。那么 *T.super.name* 正好被当成它已经是表达式 *((S)T.this).name*。

因此，表达式 *T.super.name* 可以访问在 *S* 命名的类中可见的字段命名 *name*，即使该字段通过在 *T* 命名的类中声明一个字段命名 *name* 隐藏了。

如果当前的类不是类 *T* 的一个内部类或类 *T* 本身，那就会出现编译时错误。

## 15.12 内存调用表达式

方法调用表达式用于调用类或实例方法。

*MethodInvocation*:

*MethodName* ( *ArgumentList<sub>opt</sub>* )

*Primary* . *NonWildTypeArguments<sub>opt</sub>* *Identifier* ( *ArgumentList<sub>opt</sub>* )



```

super . NonWildTypeArgumentsopt Identifier ( ArgumentListopt )
ClassName . super . NonWildTypeArgumentsopt Identifier ( ArgumentListopt )
TypeName . NonWildTypeArguments Identifier ( ArgumentListopt

```

来自第 15.9 节的 *ArgumentList* 的定义在这里为方便起见进行了重复:

*ArgumentList*:

*Expression*

*ArgumentList* , *Expression*

相对于解析一个字段名称, 在编译时解析一个方法名称是更加复杂的, 因为可能存在着方法重载。相对于访问字段, 在运行时调用方法也是更加复杂的, 因为可能存在实例方法重载。

确定由方法调用表达式调用的方法涉及几个步骤。下面 3 节描述了方法调用的编译时处理: 方法调用表达式的类型的确定在第 15.12.3 节描述。

### 15.12.1 编译时步骤 1: 确定要搜索的类或接口

在编译时, 处理方法调用的第一步是计算出要调用的方法的名称及哪个类或接口用于检查那个名称的方法的定义。根据左括号前面的形式, 有几种要考虑的情形:

- 如果形式是 *MethodName*, 那么有 3 种子情形:
  - ◆ 若它是一个简单的名称, 也就是说, 只是一个 *Identifier*, 那么方法的名称就是 *Identifier*。如果 *Identifier* 用那个名称出现在可视方法声明的作用域 (6.3 节) 中, 那么必须有一种该方法是其一个成员的封闭类型声明。令 *T* 是一个最内部的这种类型声明。要搜索的类或接口是 *T*。
  - ◆ 若它是一个形如 *TypeName.Identifier* 的限定名称, 那么方法的名称是 *Identifier*, 并且要搜索的类是 *TypeName* 命名的类。如果 *TypeName* 是一个接口而不是一个类的名称, 那么编译时错误就发生了, 因为这种形式只可以调用 *static* 方法, 并且接口没有 *static* 方法。
  - ◆ 在所有情形中, 限定名称具有形式 *FieldName.Identifier*, 那么方法的名称是 *Identifier*, 或者要搜索的类或接口是由 *FieldName* 命名的声明类型为 *T* 的字段, 如果 *T* 是一个类或接口类型, 或者如果 *T* 是一个类型变量, 则为 *T* 的上限。
- 如果形式是 *Primary.NonWildTypeArguments<sub>opt</sub>Identifier*, 那么方法的名称是 *Identifier*。令 *T* 是 *Primary* 表达式的类型: 如果 *T* 是一个变量或接口类型, 那么要搜索的类或接口是 *T*, 而如果 *T* 是一个类型变量的话, 则为 *T* 的上限。
- 如果形式是 *super.NonWildTypeArguments<sub>opt</sub>Identifier*, 那么方法的名称是 *Identifier*, 并且要搜索的类是其声明包含方法调用的类的超类。令 *T* 是立即封闭方法调用的类型声明。如果下面的任何情形之一发生, 它就是一个编译时错误:
  - ◆ *T* 是类 *Object*。
  - ◆ *T* 是一个接口。

- 如果形式是 *ClassName.super.NonWildTypeArguments<sub>opt</sub>Identifier*, 那么方法的名称就是 *Identifier*, 并且要搜索的类是 *ClassName* 表示的类 *C* 的超类。如果 *C* 不是当前类的一个词法上封闭的类, 它就是一个编译时错误。如果 *C* 是类 *Object*, 那么它就是一个编译时错误。令 *T* 是立即封闭方法调用的类型声明。如果下面的任何情形之一发生, 它就是一个编译时错误:
  - ◆ *T* 是类 *Object*。
  - ◆ *T* 是一个接口。
- 如果形式是 *TypeName.NonWildTypeArguments Identifier*, 那么方法的名称是 *Identifier*, 并且要搜索的类是 *TypeName* 表示的类 *C*。如果 *TypeName* 是一个接口而不是一个类的名称, 那么编译时错误就发生, 因为这种形式可以只调用 *static* 方法, 并且接口没有 *static* 方法。

### 15.12.2 编译时步骤 2: 确定方法签名

号召书法家为他们的意见签名……

——阿加莎·克里斯蒂,《斯泰尔斯庄园奇案》(1920)第 11 章

第二个步骤搜索成员方法的前一个步骤中确定的类型中。本步骤使用方法的名称和参数表达式的类型来查找方法, 这些方法都是可访问和可应用的, 也就是说, 声明可以在给定参数上正确调用。可能有更多的这样的方法, 在这种情形中选择最特定的方法。最特定方法的描述符(签名加返回类型)是在运行时使用的方法, 用于执行方法指派。

如果方法可通过求子类型(15.12.2.2 节)进行应用, 通过方法调用转换(15.12.2.3 节)进行应用, 那么这个方法就是可应用的, 否则它是一个可应用的可变元数方法(15.12.2.4 节)。

确定可应用性的过程首先要确定潜在的可应用方法(15.12.2.1 节)。该过程的其余部分被分为 3 个阶段。

---

分成阶段的目的是要确保与较老的 Java 编程语言版本兼容。

---

第一个阶段(15.12.2.2 节)在不允许装箱或拆箱转换, 或者使用可变元数方法调用的情况下执行重载解析。如果在这个阶段没有找到任何可应用的方法, 那么处理就继续到第二阶段。

---

这保证了在较老的语言版本中有效的任何调用不被当成不明确的, 因为引入可变元数方法、隐式装箱和/或拆箱。

---

第二阶段（15.12.2.3 节）执行重载解析，同时允许装箱和拆箱，但仍然排除使用可变元数方法调用。如果在这个阶段没有找到可应用的方法，那么处理就继续到第三阶段。

### 讨论

这确保了如果存在一个可应用的固定元数方法的话，可变元数方法就永远不会被调用。

第三阶段（15.12.2.4 节）允许重载，以便与可变元数方法、装箱和拆箱结合。

确定一个方法是否是可应用的将——在泛型方法（8.4.4 节）的情形中——要求确定实际的类型参数。实际的类型参数可以显式或隐式地进行传递。如果它们隐式进行传递，那么它们需要从参数表达式的类型中进行推断（15.12.2.7 节）。

如果在可应用性测试的 3 个阶段期间已经确定了几种可应用方法，那么就选择最特定的那个方法，如第 15.12.2.5 节所指出。有关详细信息，请参阅下面的子节。

#### 15.12.2.1 确定潜在的可应用方法

当且仅当下面的所有情况都为真时，成员函数潜在可应用到方法调用：

- 成员的名称等于方法调用中的方法的名称。
- 对于方法调用出现所在的类或接口，这个成员是可访问的（6.6 节）。
- 成员的元数少于或等于方法调用的元数。
- 如果成员是元数为  $n$  的可变元数方法，那么方法调用的元数大于或等于  $n-1$ 。
- 如果成员是一个元数为  $n$  的固定元数方法，那么方法调用的元数等于  $n$ 。
- 如果方法调用包括显式类型参数，并且成员是一个泛型方法，那么实际类型参数的数量等于形式类型参数的数量。

上面的子句暗示着对于提供显式类型参数的调用，非泛型方法可能是潜在可应用的。的确，可以证明是可应用的。在这样的一个情形中，类型参数将简单地被忽略。

此规则源自可替代性的兼容性和原理的一些问题。因为接口或超类可能对它们的子类型独立地进行泛型化，所以我们可以用一个非泛型的方法重写一个泛型的方法。但重写（非泛型）方法必须可应用于泛型方法的调用，包括显式传递参数的调用。否则，子类型不会替换它的泛型化超类型。

成员方法在方法调用时是否是可访问的取决于成员的声明中的访问修饰符（`public`、`protected` 或 `private`），或者取决于方法调用出现在哪里。

由编译时步骤 1（15.12.1 节）确定的类或接口被搜索，以便找出对于此方法调用潜在可应用的所有成员方法：在本搜索中包括了继承自超类或超接口的成员。

此外，如果在左括号之前，方法调用有一个形如 *Identifier* 的 *MethodName*，那么搜索过程也检查所有的方法，这些方法（a）由方法调用发生所在的编译单元（7.3 节）中单静态导入声明（7.5.3 节）和静态按需导入声明（7.5.4 节）导入，和（b）在方法调用出现的

地方不被屏蔽 (6.3.1 节)。

如果搜索没有产生至少一个潜在可应用的方法, 那么编译时错误就发生。

#### 15.12.2.2 阶段 1: 确定可通过求子类型应用的匹配元数方法

令  $m$  是一个潜在可应用方法 (15.12.2.1 节), 令  $e_1, \dots, e_n$  是方法调用的实际参数表达式, 并且令  $A_i$  是  $e_i$  的类型 ( $1 \leq i \leq n$ )。那么:

- 如果  $m$  是一个泛型方法, 那么令  $F_1 \dots F_n$  是  $m$  的形式参数的类型, 令  $R_1 \dots R_p$  ( $p \geq 1$ ) 是  $m$  的形式参数类型, 并令  $B_l$  是  $R_l$  的声明界限,  $1 \leq l \leq p$ 。那么:
  - 若方法调用没有提供显式的类型参数, 那么令  $U_1 \dots U_p$  是针对  $m$  的调用进行推断的实际类型参数 (15.12.2.7 节), 并针对类型为引用类型的每个实际参数表达式  $e_i$  使用一组包含约束  $A_i <: F_i$  的初始约束 ( $1 \leq i \leq n$ )。
  - 否则令  $U_1 \dots U_p$  是在方法调用中给定的显式类型参数。

然后令  $S_i = F_i [R_l = U_l, \dots, R_p = U_p]$  ( $1 \leq i \leq n$ ) 是针对  $m$  的形式参数推断的类型。

- 否则, 令  $S_1 \dots S_n$  是  $m$  的形式参数的类型。

方法  $m$  可通过求子类型进行应用, 当且仅当下面的条件成立:

- 对于  $1 \leq i \leq n$ , 为二者之一:
  - $A_i$  是  $S_i$  ( $A_i <: S_i$ ) 一个子类型 (4.10 节), 或者
  - 通过未检查转换 (5.1.9 节)  $A_i$  可以转换到某个类型  $C_i$ , 并且  $C_i <: S_i$ 。
- 如果  $m$  是一个如上面描述的泛型方法, 那么  $U_l <: B_l [R_l = U_l, \dots, R_p = U_p]$ ,  $1 \leq l \leq p$ 。

如果发现没有可通过求子类型应用的方法, 那么可应用方法的搜索就继续到第二个阶段 (15.12.2.3 节)。否则, 在可通过求子类型的访问之间选择最特定的方法 (15.12.2.5 节)。

#### 15.12.2.3 阶段 2: 确定可由方法调用转换应用的匹配元数方法

令  $m$  是一个潜在可应用方法 (15.12.2.1 节), 令  $e_1, \dots, e_n$  是方法调用的实际参数表达式, 并令  $A_i$  是  $e_i$  的类型,  $1 \leq i \leq n$ 。那么:

- 如果  $m$  是一个通用方法, 那么令  $F_1 \dots F_n$  是  $m$  的形式参数的类型, 令  $R_1 \dots R_p$  ( $p \geq 1$ ) 是  $m$  的形式类型参数, 并令  $B_l$  是  $R_l$  的声明界限,  $1 \leq l \leq p$ 。那么:
  - 若方法调用没有提供显式参数, 那么令  $U_1 \dots U_p$  是针对  $m$  的此调用进行推断 (15.12.2.7 节) 的实际类型参数, 及使用一组包括约束  $A_i <: F_i$  ( $1 \leq i \leq n$ ) 的初始约束。
  - 否则, 令  $U_1 \dots U_p$  是方法调用中给出的显式类型参数。

然后令  $S_i = F_i [R_l = U_l, \dots, R_p = U_p]$  ( $1 \leq i \leq n$ ) 是针对  $m$  的形式参数推断的类型。

- 否则, 令  $S_1 \dots S_n$  是  $m$  的形式参数的类型。

当且仅当下面的条件满足时, 方法  $m$  可由方法调用转换应用:

- 对于  $1 \leq i \leq n$ ,  $e_i$  的类型  $A_i$  可以通过方法调用转换 (5.3 节) 转换到  $S_i$ 。
- 如果  $m$  是所上面所描述的泛型方法, 那么  $U_l <: B_l [R_l = U_l, \dots, R_p = U_p]$ ,  $1 \leq l \leq p$ 。

如果没有找可通过方法调用转换进行应用的方法, 那么可应用方法的搜索就继续到第 3 阶段 (15.12.2.4 节)。否则, 就在方法调用转换可应用的方法之间选择最特定的方法 (15.12.2.5 节)。



## 15.12.2.4 阶段 3: 确定可应用的可变元数方法

令  $m$  是一个具有可变元数的潜在可应用方法 (15.12.2.1 节), 令  $e_1, \dots, e_k$  是方法调用的实际参数表达式, 并令  $A_i$  是  $e_i$  的类型,  $1 \leq i \leq k$ 。那么:

- 如果  $m$  是一个泛型方法, 那么令  $F_1 \dots F_n$  (其中  $1 \leq n \leq k+1$ ) 是  $m$  的形式参数的类型, 其中对于某个类型  $T$ , 有  $F_n = T[]$ , 并令  $R_1 \dots R_p$  ( $p \geq 1$ ) 是  $m$  的形式类型参数, 令  $B_l$  是  $R_l$  的声明限界 ( $1 \leq l \leq p$ )。那么:
  - 若方法调用没有提供显式的类型参数, 那么令  $U_1 \dots U_p$  是针对  $m$  的此调用推出 (15.12.2.7 节) 的实际类型参数, 所使用的是一组包含约束  $A_i << F_i$  ( $1 \leq i < n$ ) 和约束  $A_j \leq T$  ( $n \leq j \leq k$ ) 的初始约束。
  - 否则, 令  $U_1 \dots U_p$  是方法调用中给出的显式类型参数。

然后令  $S_i = F_i [R_1 = U_1, \dots, R_p = U_p]$   $1 \leq i \leq n$  是针对  $m$  的形参推出类型。

- 否则, 令  $S_1 \dots S_n$  (其中  $n \leq k+1$ ) 是  $m$  的形参的类型。

当且仅当下面所有 3 个条件都成立时, 方法  $m$  是可应用的可变元数方法:

- 对于  $1 \leq i < n$ ,  $e_i$  的类型  $A_i$  可通过方法调用转换转换成  $S_i$ 。
- 如果  $k \geq n$ , 那么对于  $n \leq i \leq k$ ,  $e_i$  的类型  $A_i$  可以通过方法调用转换到  $S_n$  的元素类型。
- 如果  $m$  是上面描述的泛型方法, 那么  $U_l <: B_l [R_1 = U_1, \dots, R_p = U_p]$ ,  $1 \leq l \leq p$ 。

如果没有找到可应用的可变元数方法, 就会发生一个编译时错误。否则, 在可应用的可变元数访问之中选择最特定的方法 (15.12.2.5 节)。

## 15.12.2.5 选择最特定的方法

如果多个成员方法对于一个方法调用都是可访问和可应用的, 那就需要选择一个方法来提供运行时方法指派的描述符。Java 编程语言使用选择最特定方法的规则。

坦白讲, 如果可以将第一个方法处理的任何调用传递给其他调用而不会发生运行时错误, 那么一个方法就比另一个方法更加特定。

一个名为  $m$  的固定元数成员方法比相同名称和元数的另一个成员方法更加特定, 前提是所有下面的条件都要成立:

- 第一个成员方法的参数的声明类型是  $T_1, \dots, T_n$ 。
- 其他方法的参数的声明类型是  $U_1, \dots, U_n$ 。
- 如果第二个方法是泛型的, 那么令  $R_1 \dots R_p$  ( $p \geq 1$ ) 是它的形式类型参数, 令  $B_l$  是  $R_l$  的声明限界 ( $1 \leq l < p$ ), 令  $A_1 \dots A_p$  是在初始约束  $T_i << U_i$  ( $1 \leq i \leq n$ ) 下针对此调用进行推断的实际类型参数 (15.12.2.7 节), 并令  $S_i = U_i [R_1 = A_1, \dots, R_p = A_p]$  ( $1 \leq i \leq n$ ); 否则令  $S_i = U_i$  ( $1 \leq i \leq n$ )。
- 对于所有的  $j$  (从 1 至  $n$ ),  $T_j <: S_j$ 。
- 如果第二个方法是如上面所描述的泛型方法, 那么  $A_l <: B_l [R_1 = A_1, \dots, R_p = A_p]$  ( $1 \leq l \leq p$ )。

此外, 如果下面条件之一成立, 一个名为  $m$  的可变元数方法比相同名称的另一个可变元数成员方法更加特定:

- 一个成员方法有  $n$  个参数, 另一个成员方法有  $k$  个参数, 其中  $n \geq k$ 。第一个成员方

法的参数的类型是  $T_1, \dots, T_{n-1}, T_n[]$ , 另一个方法的参数的类型是  $U_1, \dots, U_{k-1}, U_k[]$ 。如果第二个方法是泛型的, 那么令  $R_1 \dots R_p$  ( $p \geq 1$ ) 是它的形式类型参数, 令  $B_l$  是  $R_l$  的声明限界 ( $1 \leq l \leq p$ ), 令  $A_1 \dots A_p$  是在初始约束  $T_i < U_i$  ( $1 \leq i \leq k-1$ )、 $T_i < U_k$  ( $k \leq i \leq n$ ) 下针对此调用推断的实际类型参数 (15.12.2.7 节)。并令  $S_i = U_i [R_1 = A_1, \dots, R_p = A_p]$  ( $1 \leq i \leq k$ ); 否则, 令  $S_i = U_i$  ( $1 \leq i \leq k$ )。那么:

- ◆ 对于所有的  $j$  从 1 至  $k-1$ ,  $T_j < S_j$ , 并且
- ◆ 对于所有  $j$  从  $k$  至  $n$ ,  $T_j < S_k$ , 并且
- ◆ 如果第二个方法是一个如上面描述的泛型方法, 那么  $A_l < B_l [R_1 = A_1, \dots, R_p = A_p]$ ,  $1 \leq l < p$ 。
- 一个成员方法有  $k$  个参数, 另一个成员方法有  $n$  个参数, 其中  $n \geq k$ 。第一个方法的参数的类型是  $U_1, \dots, U_{k-1}, U_k[]$ , 另一个方法的参数的类型是  $T_1, \dots, T_{n-1}, T_n[]$ 。如果第二个方法是泛型的, 那么令  $R_1 \dots R_p$  ( $p \geq 1$ ) 是它的形式类型参数, 令  $B_l$  是  $R_l$  的声明限界 ( $1 \leq l \leq p$ ), 令  $A_1 \dots A_p$  是在初始条件  $U_i < T_i$  ( $1 \leq i \leq k-1$ )、 $U_k < T_i$  ( $k \leq i \leq n$ ) 下针对此调用推出的实际类型参数 (15.12.2.7 节), 并令  $S_i = T_i [R_1 = A_1, \dots, R_p = A_p]$  ( $1 \leq i \leq n$ ); 否则, 令  $S_i = T_i$  ( $1 \leq i \leq n$ )。那么:
  - ◆ 对于所有的  $j$  从 1 至  $k-1$ ,  $U_j < S_j$ , 并且
  - ◆ 对于所有  $j$  从  $k$  至  $n$ ,  $U_k < S_j$ , 并且
  - ◆ 如果第二个方法是如前面所描述的方法, 那么  $A_l < B_l [R_1 = A_1, \dots, R_p = A_p]$  ( $1 \leq l \leq p$ )。

上面的条件是一个方法可能比另一个方法更加特定的惟一条件。

当且仅当  $m_1$  比  $m_2$  更加特定, 并且  $m_2$  没有比  $m_1$  更加特定, 方法  $m_1$  是严格上比另一个方法  $m_2$  更加特定的。

如果方法是可访问和可应用的, 并且没有其他方法是可应用和访问的, 也没有严格意义上更加特定的其他方法, 我们就说这个方法对于方法调用来说是最大特定的。

如果正好有一个最大特定方法, 那么那个方法事实上就是最大特定方法。它一定要比可应用的任何其他访问方法更加特定。然后它遵从第 15.12.3 节中描述的某些未来编译时检查。

没有什么方法是最特定的情况也可能出现, 因为有两个或多个是最大特定的方法。在这种情况下:

- 如果所有的最大特定方法有重载等价 (8.4.2 节) 签名, 那么:
  - ◆ 若正好有一个最大特定方法没有被声明为 `abstract`, 那么它就是最特定方法。
  - ◆ 否则, 如果最大特定方法被声明为 `abstract`, 并且所有最大特定方法的签名都有相同的签名 (4.6 节), 那么最特定方法是在具有最特定返回类型的最大特定方法的子集之间任意进行选择。但当且仅当在每个最大特定的方法的 `throws` 子句中声明检查异常及其消除时, 最大特定方法才被考虑成抛出检查异常。
- 否则, 我们说方法调用是不明确的, 并会发生编译时错误。

#### 15.12.2.6 方法结果和抛出类型

- 选择的方法的返回类型如下进行确定:



- ◆ 如果被调用的方法用 `void` 的返回类型进行声明, 那么结果是 `void`。
- ◆ 否则, 如果未被检查的转换对于使方法可应用是必需的, 那么结果类型是方法的声明返回类型的消除。
- ◆ 否则, 如果被调用的方法是泛型的, 那么对于  $i \leq n$ , 令  $F_i$  是方法的形式类型参数, 令  $A_i$  是针对方法调用推出的实际类型参数, 并令  $R$  是正被调用的方法的声明返回类型。结果类型是通过应用捕获转换 (5.1.10 节) 到  $R[F_1 := A_1, \dots, F_n := A_n]$  而获得的。
- ◆ 否则, 结果类型是通过应用捕获转换 (5.1.10 节) 到方法声明中给出的类型而获得的。

选择的方法的 `throws` 子句的异常类型如下进行确定:

- 如果未被检查的转换对于使方法可应用是必需的, 那么 `throws` 子句就由方法的声明 `throws` 子句中的类型消除 (4.6 节) 组成。
  - 否则, 如果被调用的方法是泛型的, 那么对于  $1 \leq i \leq n$ , 令  $F_i$  是方法的形式类型参数, 令  $A_i$  是针对方法调用推出的实际类型参数, 并令  $E_j$  ( $1 \leq j \leq m$ ) 是被调用的方法的 `throws` 子句中声明的异常类型。 `throws` 子句将类型包括到  $E_j[F_1 := A_1, \dots, F_n := A_n]$ 。
  - 否则, `throws` 子句的类型是方法声明中给出的类型。
- 当且仅当下面条件之一时, 方法调用表达式可以抛出一种异常类型:
- 要调用的方法的形式为 `Primary.Identifier`, 并且 `Primary` 表达式可以抛出 `E`; 或者
  - 参数列表的一些表达式可以抛出 `E`; 或者
  - `E` 在被调用的方法的类型的 `throws` 子句中列出。

#### 15.12.2.7 基于实际参数推断类型参数

在本节, 我们描述推断方法和构造函数调用的过程。当测试方法 (或构造函数) 的可应用性时, 这个过程作为子例程调用 (15.12.2.2 节至 15.12.2.4 节)。

类型推论的过程本质上是复杂的。因此, 在研究详细的内容前, 给出这个过程的非正式概述是有用的。

推论从一组初始的约束开始。通常, 约束要求在给定声明的形式参数类型的情况下, 实际参数的静态已知类型是可接受的。下面我们讨论“可接受”的含义。

给定这些初始约束, 就可以在方法或构造函数的形式类型参数上派生出一组子类型和/或相等约束。

接下来, 我们必须尝试和查找满足类型参数上的约束的解决方案。作为第一个近似解决方案, 如果类型参数是由相等约束约束的, 那么该约束就给出它的解决方案。请谨记, 这种约束可能让一种类型参数等于另一种类型参数, 并且只有在所有类型变量上整组约束被解析的情况下, 我们才有一个解决方案。

超类型约束  $T :> X$  暗示解决方案是  $X$  的超类型之一。在  $T$  上给定几个这样的约束, 我

们可以对每个约束暗示的几个超类型的集合进行相交，因为类型参数必须是所有它们的一个成员。然后我们可以选择在该交集的最特定类型。

计算交集比我们可能最先实现的更加复杂。假设类型参数被约束到泛型类型的两个不同调用的子类型，比方说 `List<Object>` 和 `List<String>`，交集操作可能产生 `Object`。但一个更加复杂的分析产生了一个包含 `List<?>` 的集合。同样，如果一个类型参数 `T` 被约束到两个未相关的接口 `I` 和 `J` 的一个超类型，我们可以推断 `T` 必须是 `Object`，或者我们可以获得 `I & J` 的一个更紧的限界。本节更加详细地讨论了这些问题。

在本节中我们使用了下面的符号规则：

- 类型表达式使用字母 *A*、*F*、*U* 和 *W* 表示。字母 *A* 只用于表示实际参数的类型，*F* 只用于表示形参的类型。
- 类型参数使用字母 *S* 和 *T* 表示。
- 参数化类型的参数使用字母 *X*、*Y* 表示。
- 泛型类型声明使用字母 *G* 和 *H* 表示。

推论从一组形如  $A < F$ 、 $A = F$  或  $A > F$  的初始约束开始，其中  $U < V$  指出类型 *U* 可通过方法调用转换（5.3 节）转换到类型 *V*，并且  $U > V$  指出类型 *V* 或通过方法调用转换转换到类型 *U*。

简单来说，约束可以是  $A < F$  的形式——只要求实际参数类型是形式参数类型的子类型。但实际是更加复杂的。如前面所讨论，方法可应用性测试包括高达 3 个阶段：这是出于兼容性原因所需的。每个阶段略微地施加了不同约束。如果方法是可通过求子类型（15.12.2.2 节）应用的，那么约束就确实对约束求子类型。如果方法是可通过方法调用转换（15.12.2.3 节）应用的，约束就暗示实际类型可通过方法调用转换转换到形式类型。此情形与类型第 3 个阶段（15.12.2.4 节）类似，但约束的准确形式由于可变的元数而不同。

然后这些约束被缩减到一组更加简单的约束，形式为  $T > X$ 、 $T = X$  或  $T < X$ ，其中 *T* 是方法的一个类型参数。这个缩减是通过下面给出的过程获得的。

初始约束可能是不令人满意的：我们说推论是过分约束的。在该情形中，我们不一定要在类型参数上派生不可满足的约束。相反，我们可以推断调用的类型参数，但一旦我们用实际类型参数替代形式类型参数，可应用性测试可能失败，因为在给定替换形式下，实际参数类型是不可接受的。

另一种方法是让类型推论本身在这样一些情形中失败。编译器可能选择这样做，前提是效果与这里指定的等同。

给定形如  $A \ll F$ 、 $A = F$  或  $A \gg F$  的约束：

- 如果  $F$  没有包括类型参数  $T_j$ ，那就显示  $T_j$  上没有约束。
- 否则， $F$  包含一个类型参数  $T_j$ 。
  - ◆ 若  $A$  的类型是 `null`，那就暗示在  $T_j$  上没有约束。
  - ◆ 否则，如果约束有形式  $A \ll F$ 
    - ◇ 如果  $A$  是一个基本类型，那么  $A$  通过装箱转换转换到引用类型  $U$ ，并且该算法被递归地应用到约束  $U \ll F$ 。
    - ◇ 否则，如果  $F = T_j$ ，那就暗示约束  $T_j \supseteq A$ 。
    - ◇ 如果  $F = U[]$ ，其中类型  $U$  包括  $T_j$ ，那么如果  $A$  是一个数组类型  $V[]$  或一个类型变量，其上限是一个数组类型  $V[]$ （其中  $V$  是一个引用类型），那么这个算法就递归地应用到约束  $V \ll U$ 。

### 讨论

这从数组类型之间的协变子类型推出。在这种情况下，约束  $A \ll F$  意味着  $A \ll U[]$ 。因此  $A$  一定是一个数组类型  $V[]$  或类型变量（其中上限是一个数组类型  $V[]$ ）——否则，关系  $A \ll U[]$  可能不会保持成立。所以，可以得到  $V[] \ll U[]$ 。由于数组求子类型是协变的，所以它必须是  $V \ll U$  的情形。

- ◇ 如果  $F$  具有形式  $G \langle \dots, Y_{k-1}, U, Y_{k+1}, \dots \rangle$ ， $1 \leq k \leq n$ ，其中  $U$  是一个包含  $T_j$  的类型表达式，那么如果  $A$  具有形如  $G \langle \dots, X_{k-1}, V, X_{k+1}, \dots \rangle$  的子类型，则这个算法就被递归地应用到约束  $V = U$ 。

### 讨论

为简单起见，假定  $G$  只接受一个类型参数。如果被检查的方法调用要是可应用的，那么它就必须是这样的情形： $A$  是  $G$  的某个调用的子类型。否则， $A \ll F$  将不会是真的。

换言之， $A \ll F$ （其中  $F = G \langle U \rangle$ ）暗示着对于某个  $V$  有  $A \ll G \langle V \rangle$ 。现在，由于  $U$  是一个类型表达式（因此， $U$  不是一个通配符类型参数），它必须是  $U = V$  的情形，这是根据普通参数化类型调用的不变性而来。

上面的公式只概述了具有任何数量的类型参数的泛型的原因。

- ◇ 如果  $F$  有形式  $G \langle \dots, Y_{k-1} ? \text{extends } U, Y_{k+1}, \dots \rangle$ ，其中  $U$  包括  $T_j$ ，那么如果  $A$  有下面之一的超类型：
  - $G \langle \dots, X_{k-1}, V, X_{k+1}, \dots \rangle$ ，其中  $V$  是一个类型表达式。然后该算法被递归地应用到约束  $V \ll U$ 。

### 讨论

再提一下，让我们尽可能简单地论及这些内容，并且只考虑  $G$  只有一个类型参数的情形。

在本例中,  $A \ll F$  意味  $A \ll G \text{? extends } U$ 。像上面那样, 必须是这样的情形:  $A$  是  $G$  的某个调用的子类型。但  $A$  现在可能是  $G \langle V \rangle$  或  $G \text{? extends } V$  或  $G \text{? super } V$  的一个子类型。我们依次检查这些情形。第一种变化是通过上面的子项进行描述的 (一般化到多个参数)。因此, 我们有  $A = G \langle V \rangle \ll G \text{? extends } U$ 。求通配符的子类型的规则暗示着  $V \ll U$ 。

□  $G \langle \dots, X_{k-1} \text{? extends } V, X_{k+1}, \dots \rangle$ 。那么这种算法就被递归地应用到约束  $V \ll U$ 。

扩展上面的分析, 我们有  $A = G \text{? extends } V \ll G \text{? extends } U$ 。对通配符求子类型的规则再次暗示着  $V \ll U$ 。

□ 否则, 在  $T_j$  上没有暗示任何约束。

这里, 我们有  $A = G \text{? super } V \ll G \text{? extends } U$ 。通常, 在这种情形下, 我们不能总结出什么。但这不一定是一个错误。可能是  $U$  被最终被推断为 `Object`, 在这种情形中, 调用可能真的是有效的。因此, 我们只要避免在  $U$  上布置任何约束。

◇ 如果  $F$  具有形式  $G \langle \dots, Y_{k-1} \text{? super } U, Y_{k+1}, \dots \rangle$ , 其中  $U$  包括  $T_j$ , 那么如果  $A$  具有下面之一的超类型:

□  $G \langle \dots, X_{k-1}, V, X_{k+1}, \dots \rangle$ 。那么此算法没有被递归应用到约束  $V \gg U$ 。

通常, 我们只考虑  $G$  具有单类型参数的情形。

在本例中,  $A \ll F$  意味着  $A \ll G \text{? super } U$ , 如上面一样, 它必须是这样的情形:  $A$  是  $G$  的一些调用的子类型。现在可以是  $G \langle V \rangle$  或  $G \text{? extends } V$  或  $G \text{? super } V$  的一个子类型。我们依次检查这些情形。第一种变化是通过正上面的子项进行描述的 (一般化到多个参数)。因此, 我们有  $A = G \langle V \rangle \ll G \text{? super } U$ 。对通配符求子类型的规则暗示  $V \gg U$ 。

□  $G \langle \dots, X_{k-1}, \text{? super } V, X_{k+1}, \dots \rangle$ 。那么此算法被递归地应用到约束  $V \gg U$ 。

我们有  $A = G \text{? super } V \ll G \text{? super } U$ 。对下限通配符的求子类型的规则再次暗示了  $V \gg U$ 。

□ 否则, 暗示在  $T_j$  上没有任何约束。

### 讨论

这里, 我们有  $A = G < ? \text{ extends } V > < < G < ? \text{ super } U >$ 。通常, 我们不能在这种情形中总结出任何东西。但这不一定是个错误。它可能是  $U$  被最终被推断为 `null` 类型, 在这种情形中调用可能确实是有效的。因此, 我们只要避免在  $U$  上布置任何约束。

◇ 否则, 暗示在  $T_j$  上没有任何约束。

◆ 否则, 如果约束具有形式  $A = F$ 。

### 讨论

这样的一种约束从未是初始约束的一部分。但当算法重复时, 它可以产生。我们已经在上面看到这种情形发生——当约束  $A < < F$  将两个参数化类型相关起来时, 如  $G < V > < < G < U >$  中一样。

◆ 如果  $F = T_j$ , 那么暗示着约束  $T_j = A$ 。

◆ 如果  $F = U[]$  (其中类型  $U$  调用  $T_j$ ), 那么如果  $A$  是一个数组类型  $V[]$  或者一个类型变量, 该变量具有一个数组类型  $V[]$  的上限 (其中  $V$  是一个引用类型), 那么此算法就被递归地应用到约束  $V = U$ 。

显然, 如果数组类型  $U[]$  和  $V[]$  是相同的, 那么它们的元素类型就必须是相同的。

◇ 如果  $F$  具有形式  $G < \dots, Y_{k-1}, U, Y_{k+1}, \dots >$ ,  $1 \leq k \leq n$ , 其中  $U$  是一个包括  $T_j$  的类型表达式, 那么如果  $A$  的形式是  $G < \dots, X_{k-1}, V, X_{k+1}, \dots >$ , 其中  $V$  是类型表达式, 此算法被递归地应用到约束  $V = U$ 。

◇ 如果  $F$  具有形式  $G < \dots, Y_{k-1} ? \text{ extends } U, Y_{k+1}, \dots >$ , 其中  $U$  包括  $T_j$ , 那么如果  $A$  是下面之一:

□  $G < \dots, X_{k-1}, ? \text{ extends } V, X_{k+1}, \dots >$ 。那么此算法被递归应用到约束  $V = U$ 。

□ 否则, 暗示着在  $T_j$  上没有约束。

◇ 如果  $F$  具有形式  $G < \dots, Y_{k-1}, ? \text{ super } U, Y_{k+1}, \dots >$ , 其中  $U$  包括  $T_j$ , 那么如果  $A$  是下面之一:

□  $G < \dots, X_{k-1}, ? \text{ super } V, X_{k+1}, \dots >$ 。那么此算法被递归地应用到约束  $V = U$ 。

□ 否则, 暗示在  $T_j$  上没有任何约束。

◇ 否则暗示着在  $T_j$  上没有任何约束。

◆ 否则, 如果约束具有形式  $A > > F$ 。



## 讨论

由于存在着与下限通配符相关的反变式子类型规则（形如  $G < ? \text{ super } X >$  的通配符），所以当算法递归时，这种情形发生了。

可以尝试将  $A >> F$  考虑成与  $F < < A$  相同，但推论的问题不是对称的。我们需要记住关系中那个参与者包括了要被推断的类型。

◇ 如果  $F = T_j$ ，那么暗示约束  $T_j < A$ 。

## 讨论

在推断算法的主体中我们使用这样的一些约束。但它们用在第 15.12.2.8 节中。

◇ 如果  $F = U[]$ ，其中类型  $U$  包括  $T_j$ ，那么如果  $A$  是一个数组类型  $V[]$ ，或一个具有上限为数组类型  $V[]$  的类型变量，其中  $V$  是一个引用类型，那么此算法就被递归应用到约束  $V >> U$ 。否则，暗示着在  $T_j$  上没有约束。

## 讨论

这是从数组类型之间的协变子类型关系得出的。在这种情况下，约束  $A >> F$  意味着  $A >> U[]$ 。因此， $A$  一定是一个数组类型  $V[]$  或一个类型变量，该变量的上限是数组类型  $V[]$ ——否则关系  $A >> U[]$  从未成立。可以得到  $V[] >> U[]$ ，因为求数组的子类型是协变的，所以它必须是这样的情形  $V >> U$ 。

◇ 如果  $F$  具有形式  $G < \dots, Y_{k-1}, U, Y_{k+1}, \dots >$ ，其中  $U$  是一个包括  $T_j$  的类型表达式，那么：

□ 如果  $A$  是非泛型类型的一个实例，那么暗示着在  $T_j$  上没有任何约束。

在这种情形中（再次将分析限制到一元的情形），我们有约束  $A >> F = G < U >$ 。 $A$  必须是泛型  $G$  的超类型。但由于  $A$  不是一个参数化类型，所以它不能以任何方式依赖于类型参数  $U$ 。对于是  $G$  的有效类型参数的每个  $X$ ，它是  $G < X >$  的超类型。 $U$  上没有意义的约束可以从  $A$  派生。

□ 如果  $A$  是一个泛型类型声明的调用，其中  $H$  是  $G$  或  $G$  的超类或超接口，那么：

◆ 若  $H \neq G$ ，那么令  $S_1, \dots, S_n$  是  $G$  的形式类型参数，并令  $H < U_1, \dots, U_l >$  是  $H$  的惟一调用， $H$  是  $G < S_1, \dots, S_n >$  的子类型，并令  $V = H < U_1, \dots, U_l > [S_k = U]$ ，那么，如果  $V >> F$ ，则该算法被递归应用到约束  $A >> V$ 。



## 讨论

我们这里的目标是要简化  $A$  和  $F$  之间的关系。我们针对在较简单的情形上递归地调用算法，其中实际类型参数已知为像形式一样的泛型类型声明的调用。

让我们考虑这样的情形： $H$  和  $G$  都只有一个类型参数。由于我们有约束  $A = H\langle X \rangle \gg F = G\langle U \rangle$ ，其中  $H$  不同于  $G$ ，所以它必须是这样的情形： $H$  是  $G$  的某个恰当的超类或超接口。必须有一个  $H$  的（非通配符）调用， $H$  是  $F = G\langle U \rangle$  的一个子类型。将该调用称为  $V$ 。

如果我们在约束中用  $V$  替换  $F$ ，那么将完成将相同泛型（如它发生时一样， $H$ ）的调用相关起来的目标。

我们如何计算  $V$ ？ $G$  的声明必须引入形式类型参数  $S$ ，并且必须有  $H$  的某个（非通配符）调用  $H\langle U1 \rangle$ ，它是  $G\langle S \rangle$  的超类型。然后将  $U$  的类型表达式替换  $S$  将产生  $H$  的一个（非通配符）调用  $H\langle U1 \rangle [S=U]$ ，它是  $G\langle U \rangle$  的一个超类型。例如，在最简单的实例中， $U1$  可能是  $S$ ，在这种情形中，我们有  $G\langle S \rangle \leq H\langle S \rangle$  和  $G\langle U \rangle \leq H\langle U \rangle = H\langle S \rangle [S=U] = V$ 。

可能是这样的情形： $H\langle U1 \rangle$  独立于  $S$ ——也就是， $S$  根本没有在  $U1$  中发生。但上面描述的替换仍然是有效的——在该替换中， $V = H\langle U1 \rangle [S=U] = h\langle U1 \rangle$ 。而且，在这种环境下，对于任何  $T$ ， $G\langle T \rangle \leq H\langle U1 \rangle$ ，特别地， $G\langle U \rangle \leq H\langle U1 \rangle = V$ 。

不管  $U1$  是否依赖于  $S$ ，我们已经确定类型  $V$  是  $G\langle U \rangle$  的子类型的  $H$  的调用。我们现在可以在约束  $H\langle X \rangle = A \gg V = H\langle U1 \rangle [S=U]$  上递归地调用算法。然后我们将能够将  $H$  的两种调用的类型参数关联起来，并从它们当中提取相关的约束。

- ◆ 否则，如果  $A$  的形式为  $G\langle \dots, X_{k-1}, W, X_{k+1}, \dots \rangle$ ，其中  $W$  是这个算法递归应用到约束  $W=U$  的类型表达式。

## 讨论

对于类型表达式，我们有  $A = G\langle W \rangle \gg F = G\langle U \rangle$ 。由于  $W$  是一个类型表达式（并且不是一个通配符类型参数），所以它必须是这样的情形  $W=U$ （根据参数化类型的非一致性）。

- ◆ 否则，如果  $A$  的形式是  $G\langle \dots, X_{k-1} ? \text{extends } W, X_{k+1}, \dots \rangle$ ，此算法被递归应用到约束  $W \gg U$ 。

## 讨论

对于某种类型的表达式，我们有  $A = G\langle ? \text{extends } W \rangle \gg F = G\langle U \rangle$ 。根据通配符类型的求子类型规则，必须有这样的情形  $W \gg U$ 。

- ◆ 否则，如果  $A$  的形式为  $G\langle \dots, X_{k-1}, ? \text{super } W, X_{k+1}, \dots \rangle$ ，那么此算法就被递归地应用到约束  $W \ll U$ 。

对于某个类型表达式, 我们有  $AG = \langle ? \text{ super } W \rangle \gg F = G \langle U \rangle$ 。根据通配符类型的求子类型规则, 必须有这样的情形  $W \ll U$ 。

- ◇ 否则, 暗示在  $T_j$  上没有约束。
- ◆ 否则, 没有约束应用在  $T_j$  上。
- ◆ 如果  $F$  具有形式  $G \langle \dots, Y_{k-1}, ? \text{ extends } U, Y_{k+1}, \dots \rangle$ , 其中  $U$  是包含  $T_j$  的类型表达式, 那么:
  - 如果  $A$  是一个非泛型类型的实例, 那么就暗示在  $T_j$  上没有约束。

再次将分析限制到一元的情形, 我们就有约束  $A \gg F = G \langle ? \text{ extends } U \rangle$ 。  $A$  必须是泛型类型  $G$  的一个超类型。但由于  $A$  是一个非类型化类型, 所以无论如何它不能依赖于  $U$ 。对于使得  $? \text{ extends } X$  是  $G$  的一个有效类型参数的每个  $X$ , 它是类型  $G \langle ? \text{ extends } X \rangle$  的一个超类型。  $U$  上没有有意义的约束可以从  $A$  派生。

- 如果  $A$  是泛型类型声明  $H$  的一个调用, 其中  $H$  是  $G$  或  $G$  的超类或超接口, 那么:
  - ◆ 如果  $H \neq G$ , 那么令  $S_1, \dots, S_n$  是  $G$  的形式类型参数, 并令  $H \langle U_1, \dots, U_l \rangle$  是  $H$  的惟一调用,  $H$  是  $G \langle S_1, \dots, S_n \rangle$  的超类型, 并令  $V = H \langle ? \text{ extends } U_1, \dots, ? \text{ extends } U_l \rangle [S_k = U]$ 。那么这个算法就被递归应用到约束  $A \gg V$ 。

这里我们的目标是再次简化  $A$  和  $F$  之间的关系, 并在较简单的情况下递归地调用算法, 其中实际类型参数对于调用是已知的, 调用的泛型类型与形式的泛型类型是一样的。

假定  $H$  和  $G$  只有一个类型参数。由于我们有约束  $A = H \langle X \rangle \gg F = G \langle ? \text{ extends } U \rangle$ , 其中  $H$  是不同于  $G$  的, 它必须是这样的情形:  $H$  是  $G$  的某个恰当的超类或超接口。必须有  $H \langle Y \rangle$  的调用, 使得我们必须使用  $H \langle X \rangle \gg H \langle Y \rangle$  而不是使用  $F = G \langle ? \text{ extends } U \rangle$ 。

我们如何计算  $H \langle Y \rangle$ ? 呢? 如前面一样, 注意,  $G$  的声明必须引入一个形式类型参数  $S$ , 并且必须有某个  $H$  的(非通配符)调用  $H \langle U1 \rangle$ , 它是  $G \langle S \rangle$  的一个超类型。但替换  $? \text{ extends } U$  为  $S$  通常不是合法的。要看到这一点, 假定  $U1 = T[]$ 。

相反, 我们产生了  $H$  的调用  $H \langle ? \text{ extends } U1 \rangle [S = U]$ 。在最简单的实例中,  $U1$  可能是  $S$ , 在这种情形中, 我们有  $G \langle S \rangle \ll H \langle S \rangle$  和  $G \langle ? \text{ extends } U \rangle \ll H \langle ? \text{ extends } U \rangle = H \langle ? \text{ extends } S \rangle [S = U] = V$ 。

- ◆ 否则, 如果  $A$  的形式是  $G \langle \dots, X_{k-1}, ? \text{ extends } W, X_{k+1}, \dots \rangle$ , 那么此算法就被自动应用到约束  $W \gg U$ 。

## 讨论

对于某个类型表达式  $W$ , 我们有  $A = G<? \text{ extends } W>>F = G<? \text{ extends } U>$ 。通过通配符的求子类型规则, 必须是  $W > U$  的情形。

◆ 否则, 暗示在  $T_j$  上没有约束。

◆ 如果  $F$  具有形式  $G<..., Y_{k-1}, ? \text{ super } U, Y_{k+1}, ...>$ , 其中  $U$  是涉及  $T_j$  的类型表达式表达式, 那么  $A$  是下面之一:

□ 如果  $A$  是一个非泛型类型的实例, 那么就暗示着在  $T_j$  上没有约束。

## 讨论

将分析限制到一元的情形, 我们得到约束  $A > F = G<? \text{ super } U>$ 。  $A$  必须是泛型类型  $G$  的一个超类型。但由于  $A$  不是一个参数类型, 所以从各方面来说, 它不可能依赖于  $U$ 。对于使得  $? \text{ super } X$  是有效的类型参数的每个  $X$ , 它是类型  $G<? \text{ super } X>$  的一个超类型。在  $U$  上没有某种含义的约束可以从  $A$  派生。

□ 如果  $A$  是泛型类型声明  $H$  的一个调用, 其中  $H$  是  $G$  或  $G$  的超类或超接口, 那么:

◆ 如果  $H \neq G$ , 那么令  $S_1, \dots, S_n$  是  $G$  的形式类型参数, 并令  $H<U_1, \dots, U_l>$  是  $H$  的惟一调用,  $H$  是  $G<S_1, \dots, S_n>$  的超类型, 并令  $V = H<? \text{ super } U_1, \dots, ? \text{ super } U_l>[S_k = U]$ 。然后此算法被递归地应用到约束  $A > V$ 。

这里的处理类似于  $A = G<? \text{ extends } U>$  的情形。这里我们的例子会产生一个调用  $H<? \text{ super } U_1>[S = U]$ 。

◆ 否则, 如果  $A$  的形式是  $G<..., X_{k-1}, ? \text{ super } W, \dots, X_{k+1}, ...>$ , 此算法被递归应用到约束  $W < U$ 。

## 讨论

对于某个类型表达式  $W$ , 我们有  $A = G<? \text{ super } W>>F = G<? \text{ super } U>$ 。根据通配符类型的求子类型规则, 必须是这样的情形:  $W < U$ 。

◆ 否则, 暗示着在  $T_j$  上没有约束。

这包括了在方法的形式类型参数上确定约束的过程。

注意, 这个过程没有基于类型参数的声明限界在它们之上强加任何约束。一旦实际类型

参数被推断，它们将针对形式类型参数的声明限界进行测试（作为可应用性测试的一部分）。

也请注意，从各方面而言，类型推断不会影响健壮性。如果推断的类型不是无意义的，那么调用将产生一个类型错误。类型推断算法应该被视为启发式的，设计成在实践中表现良好。如果它不能推断所需的结果，那就可能反过来使用显式的类型参数。

接下来，对于每个类型变量  $T_j$ ,  $1 \leq j \leq n$ ，暗示的相等性约束就如下进行解析：

对于每个暗示的相等性约束  $T_j = U$  或  $U = T_j$ ：

- 如果  $U$  不是方法的类型参数之一， $U$  就是针对  $T_j$  推断的类型。那么涉及  $T_j$  的所有其余约束就被重写，使得  $T_j$  用  $U$  进行替换。一定没有包括  $T_j$  的进一步相等性约束，并且处理用下一个类型参数（如果有的话）继续。
- 否则，如果  $U$  是  $T_j$ ，那么该约束就不携带任何信息，并且可能被丢弃。
- 否则，约束的形式是  $T_j = T_k$ （针对  $k \neq j$ ）。然后重写所有涉及  $T_j$  的约束，使得  $T_j$  被  $T_k$  替换，并且用下一个类型变量继续处理。

然后，对于每个其余类型变量  $T_j$ ，考虑约束  $T_j \geq U$ 。假定这些约束是  $T_j \geq U_1 \dots T_j \geq U_k$ ， $T_j$  的类型就被推断为  $\text{lub}(U_1 \dots U_k)$ ，计算如下：

对于类型  $U$ ，针对  $U$  的超类型的集合写出  $ST(U)$ ，并定义要被擦除的超类型，

$EST(U) = \{V|W \text{ in } ST(U) \text{ and } V = |W|\}$

其中  $|W|$  是  $W$  的擦除（4.6 节）。

#### 知识

计算被擦除的超类型的集合的原因是要处理这样的情形：类型变量被约束到泛型类型声明的几个不同调用的超类型。例如，如果  $T \geq \text{List}\langle \text{String} \rangle$  和  $T \geq \text{List}\langle \text{Object} \rangle$ ，只要相交集合  $ST(\text{List}\langle \text{String} \rangle) = \{\text{List}\langle \text{String} \rangle, \text{Collection}\langle \text{String} \rangle, \text{Object}\}$  和  $ST(\text{List}\langle \text{Object} \rangle) = \{\text{List}\langle \text{Object} \rangle, \text{Collection}\langle \text{Object} \rangle, \text{Object}\}$  会产生一个集合  $\{\text{Object}\}$ ，并且我们会失去跟踪这样的事实：可以安全地假定  $T$  是一个  $\text{List}$ 。

比较而言，相交  $EST(\text{List}\langle \text{String} \rangle) = \{\text{List}, \text{Collection}, \text{Object}\}$  和  $EST(\text{List}\langle \text{Object} \rangle) = \{\text{List}, \text{Collection}, \text{Object}\}$  产生  $\{\text{List}, \text{Collection}, \text{Object}\}$ ，这些最终让我们能够按如下所描述推断  $T = \text{List}\langle ? \rangle$ 。

类型参数  $T_j$  的被擦除候选集合  $EC$  是针对  $U_1 \dots U_k$  中的每个  $U$  的所有集合的交集。 $T_j$  的最小擦除候选集合是  $MEC = \{V|V \text{ in } EC, \text{ and for all } w \neq v \text{ in } EC, \text{ it is not the case that } W \leq V\}$ 。

#### 计算

因为我们正在想办法推断更准确的类型，所以我们希望过滤掉是其他候选的超类型的任何候选者。这就是计算  $MEC$  所完成的。在我们的例子中，有  $EC = \{\text{List}, \text{Collection}, \text{Object}\}$  及现在的  $MEC = \{\text{List}\}$ 。

下一步中将重新获得被推断的类型的实际类型参数。

对于是泛型类型声明的 MEC 的任何元素  $G$ ，将  $G$  的相关调用  $Inv(G)$  定义成：

$$Inv(G) = \{ V \mid 1 \leq i \leq k, V \text{ in } ST(U_i), V = G \langle \dots \rangle \}$$

### 讨论

在我们的运行示例中，MEC 仅有的泛型元素是  $List$  和  $Inv(List) = \{List \langle String \rangle, List \langle Object \rangle\}$ 。我们现在寻求发现  $List$  的一种类型参数，该  $List$  同时包含 (4.5.1.1 节)  $String$  和  $Object$ 。

这是通过下面定义的最小包含调用 ( $lci$ ) 操作的方式完成的。第一行在集合上定义了  $lci()$ ，比如  $Inv(List)$ ，将其定义成集合的元素的列表上的一个操作。接下来一行在这样的一些列表上定义了操作，将其定义成列表的元素上的成对缩减。第 3 行参数化类型上的  $lci()$  的定义，该定义依次依赖于最少包含类型参数 ( $lcta$ ) 的符号。

$lcta()$  是针对 6 个可能的情形定义的。然后， $CandidateInvocation(G)$  定义泛型  $G$  的最特定调用，泛型  $G$  包含了  $G$  的所有调用，这些调用被称为  $T_j$  的超类型。在我们针对  $T_j$  推断的限界中，这将是我们的  $G$  的候选调用。

并且令  $CandidateInvocation(G) = lci(Inv(G))$ ，其中  $lci$  (最小包含调用) 被定义了。

$$lci(S) = lci(e_1, \dots, e_n), \text{ 其中 } e_i \text{ 在 } S \text{ 中}, 1 \leq i \leq n$$

$$lci(e_1, \dots, e_n) = lci(lci(e_1, e_2), e_3, \dots, e_n)$$

$$lci(G \langle X_1, \dots, X_n \rangle, G \langle Y_1, \dots, Y_n \rangle) = G \langle lcta(X_1, Y_1), \dots, lcta(X_n, Y_n) \rangle$$

其中  $lcta()$  是定义 (假定  $U$  和  $V$  是类型表达式) 成如下的最少包含类型参数函数：

$$lcta(U, V) = U \text{ if } U = V, ? \text{ extends } lub(U, V) \text{ otherwise}$$

$$lcta(U, ? \text{ extends } V) = ? \text{ extends } lub(U, V)$$

$$lcta(U, ? \text{ super } V) = ? \text{ super } glb(U, V)$$

$$lcta(? \text{ extends } U, ? \text{ extends } V) = ? \text{ extends } lub(U, V)$$

$$lcta(? \text{ extends } U, ? \text{ super } V) = U \text{ if } U = V, ? \text{ otherwise}$$

$$lcta(? \text{ super } U, ? \text{ super } V) = ? \text{ super } glb(U, V)$$

其中  $glb()$  如在第 5.1.10 节所定义的。

### 讨论

最终，我们基于  $T_j$  的超类型的最小擦除候选集合的所有元素定义了  $T_j$  的一个限界。如果这些元素的任何元素都是泛型的，我们就使用  $CandidateInvocation()$  函数来重新获得类型参数信息。

然后，如果  $W$  是泛型的，定义  $Candidate(W) = CandidateInvocation(W)$ ，否则为  $W$ 。

那么  $T_j$  的被推断的类型是  $lub(U_1 \dots U_k) = Candidate(W_1) \& \dots \& Candidate(W_r)$ ，其中  $W_i (1 \leq i \leq r)$  是 MEC 的元素。



上面的过程产生无限大的类型是可能的。这是允许的，并且 Java 编译器必须识别这样的一些情形，并使用循环数据结构恰当地表示它们。

### 讨论

无限类型的可能性源自于对 `lub()` 的递归调用。

熟悉递归类型的读者应该注意到无限类型与递归类型是不一样的。

#### 15.12.2.8 推断未解析的类型参数

如果任何方法的类型参数没有从实际参数的类型推断，那么它们现在就像下面那样进行推断。

- 如果方法结果在它服从于转换（5.2 节）到类型  $S$  的上下文中发生，那么令  $R$  是方法的声明结果类型，并令  $R' = R[T_1 = B(T_1) \dots T_n = B(T_n)]$ ，其中  $B(T_i)$  是前一节中针对  $T_i$  推断的类型，或者如果没有类型被推断，则为  $T_i$ 。

然后，一组初始约束包含：

- 约束  $S \gg R'$ ，只要  $R$  不是 `void`；并且
- 其他的约束  $B_i[T_1 = B(T_1) \dots T_n = B(T_n)] \gg T_i$ ，其中  $B_i$  是  $T_i$  的声明限界，

它被创建和用于推断类型参数上的约束，所使用的算法是第 15.12.2.7 节。任何相等性约束被解析，然后针对形式  $T_i <: U_k$  的每个其余约束，参数  $T_i$  被推断为  $glb(U_1, \dots, U_k)$ （5.1.10 节）。

然后，还没有被推断的任何其余类型变量被推断为具有类型 `Object`。

- 否则，未解析的类型参数通过调用本节描述的过程进行推断，推断的假设是方法结果被赋予类型为 `Object` 的变量。

#### 15.12.2.9 示例

在示例程序中：

```
public class Doubler {
    static int two() { return two(1); }
    private static int two(int i) { return 2*i; }
}

class Test extends Doubler {
    public static long two(long j) {return j+j; }
    public static void main(String[] args) {
        System.out.println(two(3));
        System.out.println(Doubler.two(3)); // compile-time error
    }
}
```

对于类 `Doubler` 中的方法调用 `two(1)`，有两个名为 `two` 的可访问方法，但只有第二个就可应用的，因此那是运行时调用的一个。对于类 `Test` 中的方法调用 `two(3)`，就有两个可应用方法，但只有类 `Test` 中的一个方法是可访问的，因此那是运行时调用的一个方法（参数 3 被转换到类型 `long`）。对于方法调用 `Doubler.two(3)`，在类 `Doubler` 而不



是类 `Test` 中搜索名为 `two` 的方法；仅可应用的方法不是可访问的，因此此方法调用导致一个编译时错误。

另一个例子是：

```
class ColoredPoint {
    int x, y;
    byte color;
    void setColor(byte color) { this.color = color; }
}
class Test {
    public static void main(String[] args) {
        ColoredPoint cp = new ColoredPoint();
        byte color = 37;
        cp.setColor(color);
        cp.setColor(37); // compile-time error
    }
}
```

这里对 `setColor` 的第二个调用发生了一个编译时错误，因为在编译时不可以找到可应用的方法。文本 `37` 的类型是 `int`，`int` 可以通过方法调用转换转换到 `byte`。赋值转换可以用在变量 `color` 的初始化中，它执行常量的隐式转换从类型 `int` 到 `byte`；但这样的一种转换对于方法调用转换是不允许的。

但如果方法 `setColor` 已经被声明接受一个 `int` 而不是一个 `byte`，那么两个方法将是正确的：第一个调用会被允许，因为方法调用转换确实允许从 `byte` 到 `int` 的更宽的转换。但一个更狭窄的转换在 `setColor` 的体中会是需要的。

```
void setColor(int color) { this.color = (byte)color; }
```

#### 15.12.2.10 示例：重载不明确

考虑例子：

```
class Point { int x, y; }
class ColoredPoint extends Point { int color; }

class Test {
    static void test(ColoredPoint p, Point q) {
        System.out.println("(ColoredPoint, Point)");
    }
    static void test(Point p, ColoredPoint q) {
        System.out.println("(Point, ColoredPoint)");
    }
    public static void main(String[] args) {
        ColoredPoint cp = new ColoredPoint();
        test(cp, cp); // compile-time error
    }
}
```

这个例子在编译时产生了一个错误。问题是有两个 `test` 的声明，这两个声明是可应用和可访问的，并且没有任何一个比其他一个更加特定。因此，方法调用是不明确的。

如果 `test` 的第 3 个定义被添加了：

```
static void test(ColoredPoint p, ColoredPoint q) {
    System.out.println("(ColoredPoint, ColoredPoint)");
}
```

那么它会比其他的两个更加特定，并且方法调用会不再是不确定的。

#### 15.12.2.11 示例：返回没有考虑的类型

考虑另一个例子：

```
class Point { int x, y; }
class ColoredPoint extends Point { int color; }
class Test {
    static int test(ColoredPoint p) {
        return p.color;
    }
    static String test(Point p) {
        return "Point";
    }
    public static void main(String[] args) {
        ColoredPoint cp = new ColoredPoint();
        String s = test(cp); // compile-time error
    }
}
```

这里，方法 `test` 的最特定声明是带有一个类型 `ColoredPoint` 的参数的声明。因为方法的结果类型是 `int`，所以一个编译时错误发生了，因为一个 `int` 不可以通过赋值转换转换到 `String`。该例展示了方法的结果类型没有参与解析重载的方法，因此，第二个 `test` 方法（该方法返回一个 `string`）没有被选择，即使它有允许示例程序在没有错误的情形下编译的结果类型。

#### 15.12.2.12 示例：编译时解析

最可应用的方法是在编译时选择的：它的描述符确定了在运行时实际执行的是什么方法。如果一个新的方法被添加到一个类，那么用类的旧定义编译的源代码可能不使用新的方法，即使重新编译会导致该方法被选择。

因此，例如，考虑两个编译单元，一个针对类 `Point`：

```
package points;
public class Point {
    public int x, y;
    public Point(int x, int y) { this.x = x; this.y = y; }
    public String toString() { return toString(""); }
    public String toString(String s) {
        return "(" + x + "," + y + s + ")";
    }
}
```

另一个针对 ColoredPoint:

```
package points;
public class ColoredPoint extends Point {
    public static final int
        RED = 0, GREEN = 1, BLUE = 2;
    public static String[] COLORS =
        { "red", "green", "blue" };
    public byte color;
    public ColoredPoint(int x, int y, int color) {
        super(x, y); this.color = (byte)color;
    }
    /** Copy all relevant fields of the argument into
        this ColoredPoint object. */
    public void adopt(Point p) { x = p.x; y = p.y; }
    public String toString() {
        String s = "," + COLORS[color];
        return super.toString(s);
    }
}
```

现在考虑使用 ColoredPoint 的第 3 个编译单元:

```
import points.*;
class Test {
    public static void main(String[] args) {
        ColoredPoint cp =
            new ColoredPoint(6, 6, ColoredPoint.RED);
        ColoredPoint cp2 =
            new ColoredPoint(3, 3, ColoredPoint.GREEN);
        cp.adopt(cp2);
        System.out.println("cp: " + cp);
    }
}
```

输出是:

```
cp: (3, 3, red)
```

编码类 Test 的应用程序程序员已经期望看到单词 green, 因为实际参数 ColoredPoint 有一个 Color 字段, 并且 color 会看成是一个“相关的字段”(当然包 Points 的文档应该更加精确!)

顺便提请注意一下, adopt 的方法调用的最特定方法(当然, 仅有的可应用方法)具有一个签名, 该签名指出一个参数的方法, 并且参数的类型是 Point。该签名变成了编译器产生的类 Test 的二进制形式的一部分, 并在运行时由方法调用使用。

假设程序员报告该软件的错误, 并且 points 包的维护人员在恰当地考虑后决定通过添加一个方法到类 ColoredPoint 来纠正它:

```
public void adopt ((ColoredPoint p) {  
    adopt((Point)p): color = p.color;  
}
```

如果应用程序程序员然后运行 Test 的老的二进制文件与 ColoredPoint 的新的二进制文件, 那么输入仍然是:

```
cp: (3, 3, red)
```

因为 Test 的老的二进制文件仍然有与方法调用 cp.adopr(cp2) 有关的描述符“一个参数, 其类型是 Point; void”。如果 Test 的源代码被重新编译, 那么编译器将发现现在有两个可应用的 adopt 方法, 并且最特定方法的签名是“一个参数, 其类型是 ColoredPoint; void”, 然后运行程序将产生所需的输出。

```
cp: (3, 3, greep)
```

有了这些问题的预见, points 包的维护人员可以解决 ColoredPoint 类, 以便使用最新编译和老的代码, 方法是将防御性代码添加到老的 adopt 方法, 原因是老的代码仍然在 ColoredPoint 参数上调用它:

```
public void adopt(Point p) {  
    if (p instanceof ColoredPoint)  
        color = ((ColoredPoint)p).color;  
    x = p.x; y = p.y;  
}
```

理想地说, 当源代码所依赖的源代码改变时, 源代码应该重新进行编译。但在不同的类由不同的组织进行维护的环境中, 这始终不是有效可行的。对类演化的问题的仔细关注防御性编程可以使升级的代码更加强健。有关二进制兼容性和类型演化的详细讨论, 请参阅第 13 章。

### 15.12.3 编译时步骤 3: 选择的方法恰当吗?

如果有方法调用的一个最特定方法声明, 那么它就被称为方法调用的编译时声明。3 个进一步检查必须在编译时声明上做出:

- 如果方法调用在左括号前面有一个形如 *Identifier* 的 *MethodName*, 并且方法是一个实例方法, 那么:
  - ◆ 若调用在静态上下文 (8.1.3 节) 中出现, 编译错误就出现了 [原因是这种形式的方法不能用于在不定义 this (15.8.3 节) 的位置中调用方法]。
  - ◆ 否则, 令 *C* 是方法是其类的成员的最内封闭类。如果调用不是由 *C* 或 *C* 的内部类直接封闭的, 那么就会发生编译时错误。
- 如果方法调用在左括号之前有一个形如 *TypeName.Identifier* 的一个 *MethodName*, 或者如果在左括号前方法调用具有形式 *TypeName.NonWildTypeArguments Identifier*, 那么编译时声明应该是 *static*。如果方法调用的编译时声明是针对一个实例方法的, 那么就发生一个编译时错误 (原因是这种形式的方法调用没有指定在实际方法中可

以充当 `this` 的对象的一个引用)。

- 如果方法调用在左括号之前有形式 `super.NonWildTypeArgumentsoptIdentifier` 的方法调用, 那么:
  - ◆ 若方法是 `abstract` 时, 就会出现编译时错误。
  - ◆ 若在静态上下文中出现方法调用, 就发生编译时错误。
  - ◆ 否则, 令 `C` 是 `ClassName` 表示的类。如果调用不是直接由 `C` 或 `C` 的内部类封闭, 那么就会发生编译时错误。
- 如果方法调用的编译时声明是 `void`, 那么方法调用必须是一个顶级表达式, 也就是, 表达式语句(14.8 节)的 *Expression*, 或 `for` 语句(14.14 节)的 *ForInit* 或 *ForUpdate* 部分, 否则, 发生一个编译时错误 (原因是这样的一个方法调用没有发生值, 因此必须只用在不需要值的情形)。

然后下面的编译时信息和方法调用相关联, 以便用在运行时:

- 方法的名称
- 方法调用的限定类型 (13.1 节)
- 参数的数量和参数的类型 (按顺序)
- 结果类型或 `void`
- 调用模式, 按如下进行计算:
  - ◆ 若编译时声明具有 `static` 修饰符, 那么调用模式就是 `static`。
  - ◆ 否则, 如果编译时声明具有 `private` 修饰符, 那么调用模式是 `nonvirtual`。
  - ◆ 否则, 如果在左括号前的方法调用的形式是 `super.Identifier`, 或者形式为 `ClassName.super.Identifier`, 那么调用模式是 `super`。
  - ◆ 否则, 如果编译时声明在一个接口中, 那么调用模式是 `interface`。
  - ◆ 否则, 调用模式是 `virtual`。

如果方法调用的编译时声明不是 `void`, 那么方法调用表达式是编译时声明中指定的结果类型。

#### 15.12.4 方法调用的运行时求值

在运行时, 方法调用需要 5 个步骤。首先, 一个目标引用可以被计算。其次, 参数表达式被计算。第三, 检查要被调用的方法的可访问性。第四, 定位要被执行的方法的实际代码。第五, 创建新的激活帧, 如果有需要, 执行同步, 并且控制被传送到方法代码。

##### 15.12.4.1 计算目标引用 (如果有必要)

有几种情形要考虑, 这取决于 5 种 *MethodInvocation* (15.12 节) 的生产中的哪一种被涉及:

- 如果涉及 *MethodInvocation* (包括 *MethodName*) 的产生, 那么就有 3 种子情形:
  - ◆ 若 *MethodName* 是一个简单的名称, 也就是, 只是一个 *Identifier*, 那么就有两种子情形:
    - ◇ 如果调用模式是 `static`, 那么就没有目标引用。

- ✧ 否则, 令  $T$  是封闭类型声明, 其方法是一个成员, 并令  $n$  是一个整数, 使得  $T$  是类的第  $n$  年词法上封闭的类型声明 (8.1.3 节), 类的声明正好包含方法调用。然后目标引用是 `this` 的第  $n$  年词法上封闭的实例 (8.1.3 节)。如果 `this` 的第  $n$  个词法上封闭的实例 (8.1.3 节) 不存在, 那么它就是一个编译时错误。
  - ◆ 若 *MethodName* 是形式 *Type.Name . Identifier* 的一个限定名称, 那么就没有目标引用。
  - ◆ 若 *MethodName* 是形式 *FieldName . Identifier* 的一个限定名称, 那么就有两子情形:
    - ✧ 如果调用模式是 `static`, 那么就没有目标引用。表达式 *FieldName* 被计算, 然后被丢弃。
    - ✧ 否则, 目标引用就是表达式 *FieldName* 的值。
  - 如果 *MethodInvocation* 的第二种生产 (包括 *Primary*) 被涉及了, 那么就有两种子情形:
    - ◆ 若调用模式是 `static`, 那么就没有目标引用。表达式 *Primary* 进行求值, 但结果然后被丢弃。
    - ◆ 否则表达式 *Primary* 进行求值, 并将结果用作目标引用。
- 在任何一种情形中, 如果 *Primary* 表达式的求值突然结束, 那么就没有任何参数表达式的一部分表已经进行计算, 并且方法调用出于相应的原因突然结束。
- 如果 *MethodInvocation* 的第 3 个生产 (包括关键字 `super`) 被涉及了, 那么目标引用就是 `this` 的值。
  - 如果 *MethodInvocation* 的第 4 个生产 (*ClassName.super*) 被涉及了, 那么目标引用就是 *ClassName.this* 的值。
  - 如果 *MethodInvocation* 的第 5 个生产 (从 *TypeName.NonWildTypeArguments* 开始) 被涉及了, 那么就没有目标引用。

#### 15.12.4.2 计算参数

计算参数列表的过程是不同的, 取决于被调用的方法是否是一个固定元数的方法或者是一个可变元数方法 (8.4.1 节)。

如果被调用的方法是一个可变元数方法 (8.4.1 节)  $m$ , 那么必须有  $n > 0$  个形参。对于某个  $T$ ,  $m$  的最后一个形参必须类型  $T[]$ , 并且  $m$  必须用  $k \geq 0$  个实际参数表达式进行调用。

如果  $m$  用  $k \neq n$  个实际参数表达式进行调用, 或者, 如果  $m$  用  $k = n$  个实际参数表达式进行调用, 并且第  $k$  个参数表达式的类型不是与  $T[]$  兼容的赋值, 那么参数清单 ( $e_1, \dots, e_{n-1}, e_n, \dots, e_k$ ) 就像它被写为  $(e_1, \dots, e_{n-1}, \text{new } T[]\{e_n, \dots, e_k\})$  一样进行计算。

参数表达式 (可能像上面那样进行重写) 现在进行计算以产生参数值。每个参数值正好对应于方法的  $n$  个形参中的一个。

参数表达式 (如果有的话) 按左到右的顺序进行计算。如果任何参数表达式的计算突然地结束, 那么就没有它的右边的任何参数表达式的部分表现为已经进行计算, 并且出于相同的原因, 方法调用突然结束。计算第  $j$  个参数表达式的结果是第  $j$  个参数值 (对



于  $1 \leq j \leq n$ )。然后计算如下面描述的那样, 使用参数值继续。

#### 15.12.4.3 检测类型和方法的可访问性

令  $C$  是包含方法调用的类。令  $T$  是方法调用的限定类型 (13.1 节), 并令  $m$  是方法名称, 如在编译时所确定的一样 (15.12.3 节)。Java 编程语言的实现必须确保 (作为链接的一部分) 方法  $m$  仍旧在类型  $T$  中存在。如果不是这种情况, 那么就会发生 `NoSuchMethodError` (这是 `IncompatibleClassChangeError` 的一个子类)。如果调用模式是 `interface`, 那么实现也必须检查目标引用类型仍然实现指定的接口。如果目标引用类型仍没有实现接口, 那么就会发生一个 `IncompatibleClassChangeError`。

实现也必须确保 (在链接期间) 类型  $T$  和方法  $m$  是可访问的。对于类型  $T$ :

- 如果  $T$  在与  $C$  相同的包中, 那么  $T$  是可访问的。
- 如果  $T$  在与  $C$  不同的包中, 并且  $T$  是 `public`, 那么  $T$  是可访问的。
- 如果  $T$  在与  $C$  不同的包中, 并且  $T$  是 `protected`, 那么  $T$  是可访问的, 当且仅当  $C$  是  $T$  的一个子类。

对于方法  $m$ :

- 如果  $m$  是 `public`, 那么  $m$  是可访问的。[所有的接口成员是 `public` (9.2 节)]。
- 如果  $m$  是 `protected`, 那么当且仅当  $T$  在与  $C$  相同的包中, 或  $C$  是  $T$  或  $T$  的一个子类,  $m$  是可访问的。
- 如果  $m$  有默认的 (包) 访问, 那么当且仅当  $T$  在与  $C$  相同的包中,  $m$  是可访问的。
- 如果  $m$  是 `private`, 那么当且仅当  $C$  是  $T$ , 或  $C$  封闭  $T$ , 或  $T$  封闭  $C$ , 或  $T$  和  $C$  都被第三个类封闭,  $m$  是可访问的。

如果  $T$  或  $m$  是不可访问的, 那么会发生 `IllegalAccessException` (12.3 节)。

#### 15.12.4.4 定位要调用的方法

在我的纸杯中, 任何东西都在向上张望。

——Jim Webb, 《Paper Cup》(1967)

方法查询的策略取决于调用模式。

如果调用模式是 `static`, 那么就不需要目标引用, 并且不允许重载。类  $T$  的方法  $m$  是要调用的一个方法。

否则, 实例方法是要被调用的, 并且有一个目标引用。如果目标引用是 `null`, 那此刻就抛出 `NullPointerException`。否则, 目标引用据说引用目标对象, 并将被用作被调用方法中关键字 `this` 的值。然后考虑调用模式的其他 4 种可能性。

如果调用模式是 `nonvirtual`, 重载被禁止。类  $T$  的方法  $m$  是要调用的一个方法。

否则, 调用模式是 `interface`、`virtual` 或 `super`, 重载可能发生。动态方法查询被使用。动态查询过程从一个类开始, 像下面那样进行确定:

- 如果调用模式是 `interface` 或 `virtual`, 那么  $S$  初始是目标对象的实际运行时类  $R$ 。这种情况是真的, 即使目标对象是一个数组实例。注意, 对于调用模式 `interface`,  $R$  一定实现  $T$ ; 对于调用模式 `virtual`,  $R$  一定是  $T$  或  $T$  的子类。

- 如果调用模式是 `super`，那么  $S$  最初是方法调用的限定类型（13.1 节）。

动态方法查询使用下面的过程来搜索方法  $m$  的类  $S$ ，然后方法  $m$  的类  $S$  的超类（根据需要）。

令  $x$  是方法调用的目标引用的编译时类型。

(1) 如果类  $S$  包含名为  $m$  的非抽象方法的声明，方法调用需要相同描述符（相同参数数量，相同参数类型和相同返回类型），如编译时所确定的那样（15.12.3 节），那么：

- 如果调用模式是 `super` 或 `interface`，那么这是要调用的方法，并且过程终止。
- 如果调用模式是 `virtual`，并且  $S$  中的声明重载了（8.4.8.1 节） $x.m$ ，那么  $S$  中声明的方法就是要调用的方法并且过程终止。

(2) 否则，如果  $S$  有一个超类，此相同的查询过程就在  $S$  的位置使用  $S$  的直接超类递归地执行；要被调用的方法就是此查询过程的递归调用的结果。

上面的过程将始终找到要调用的非抽象、可访问的方法，前提是程序中的所有类和接口已经被一致地进行编译。但如果不是这种情形，那么各种错误可能发生。这些情形下的 Java 虚拟机的行为的规范是通过《The Java Virtual Machine Specification》给出的。我们注意动态查询过程（尽管在这里显式地进行描述）通常将隐式地实现，例如，作为构造的一个副作用和每个类方法指派表的使用，或用于高效指派的其他每类结构的构造。

#### 15.12.4.5 创建帧、同步和传送控件

在某些类  $S$  中的方法  $m$  已经被确定为要调用的一个方法。

现在创建一个新的激活帧，包含目标引用（如果有的话）和参数值（如果有的话），以及足够的空间来容纳局部变量和要被调用的方法的堆栈，同时实现可能需要的任何其簿记信息（堆栈指针、计数器，前一个激活指针的引用，等等）。如果没有足够的可用变量来创建这样的一个激活帧，就会抛出 `StackOverflowError`。

新创建的激活帧变成当前激活帧。其效果是将参数赋值给方法的相应新创建的参数变量值，以及使目标引用变量像 `this` 一样可用（如果有一个目标引用的话）。在每个参数值被赋值给它的相应参数变量前，它遵从与方法调用转换（5.3 节），该转换包括了任何所需的值集合转换（5.1.13 节）。

如果被调用的方法的类型的擦除在它的签名上不同于方法调用的编译时声明的类型的擦除（15.12.3 节），那么如果任何参数值是一个对象，该对象不是方法调用的编译时声明中的相应形式参数类型的擦除的子类或子接口的一个实例，那么就抛出 `ClassCastException`。

#### 讨论

作为这一情形的一个例子，考虑下面的声明：

```
class C<T> { abstract T id(T x); }  
class D extends C<String> { String id(String x) { return x; } }
```

现在给定一个调用：

```
C c = new D();
c.id(new Object()); // fails with a ClassCastException
```

正被调用的实际方法的擦除 `D.id()` 在签名方面不同于编译时方法声明的擦除 `C.id()`。前者带有一个类型为 `String` 的参数，而后者带有一个类型为 `Object` 的参数。在方法体被执行前，调用以一个 `ClassCastException` 失败了。

如果程序产生一个非受检查的警告（5.1.9 节），那么这样的一些情形就只可能发生。

通过创建桥方法，实际可以增强这些语义。在上面的例子中，下面的桥方法将在类 `D` 中创建：

```
Object id(Object x) { return id((String) x); }
```

这是实际上会由 Java 虚拟机调用的方法，以便响应上面调用的 `c.id(new Object())`，并且它将执行转换，并在需要时会失败。

如果方法 `m` 是一个 native 方法，所需的本机、实现依赖的二进制码还没有被加载，或其他的不能动态进行链接，那么就抛出 `UnsatisfiedLinkError`。

如果方法 `m` 不是 `synchronized`，那么控制就被传递给要被调用的方法 `m` 的体。

如果方法 `m` 是 `synchronized`，那么在控制被传递之前，对象就必须被锁定。不能进行进一步的进展，直到当前线程可以获得锁。如果没有目标引用，那么目标就必须被锁定；否则，类 `S` 的 `Class` 对象，方法 `m` 的类就必须被锁定。然后控制被传递给要被调用的方法的 `m` 的体。当方法体的执行完成时，而不管是正常或突然，对象自动解锁。加锁和解锁行为正像方法体被嵌入到 `synchronized` 语句一样（14.19 节）。

#### 15.12.4.6 示例：目标引用和静态方法

当目标引用被计算，且由于调用模式是 `static` 时而被丢弃时，引用就没有被检查以查看它是否是 `null`：

```
class Test{
    static void mountain() {
        System.out.println("Monadnock");
    }
    static Test favorite(){
        System.out.print("Mount ");
        return null;
    }
    public static void main(String[] args) {
        favorite().mountain();
    }
}
```

该代码输出：

```
Mount Monadnock
```

这里 `favorite` 返回 `null`，而没有抛出 `NullPointerException`。

#### 15.12.4.7 示例：计算顺序

作为实例方法调用（15.12 节）的一部分，没有一个形式表示要被调用的对象。在计算方法调用的任何参数形式的任何部分之前，此表达式呈现为全部被计算。

因此，例如，在如下示例中：

```
class Test {
    public static void main (String[] args){
        String s = "one"
        if (s.startsWith(s = "two"))
            System.out.println("oops");
    }
}
```

“`.startsWith`”前出现的 `s` 最先进行计算（在参数表达式 `s="two"` 之前）。因此，在局部变量 `s` 被更改到引用字符串 `"two"` 前，字符串 `"one"` 的引用被记为目标引用。结果，`startsWith` 方法用参数 `"two"` 调用目标对象 `"one"`，因此，调用的结果是 `false`，因此字符串 `"one"` 没有从 `"two"` 开始。可以推出测试程序没有输出 `"oops"`。

#### 15.12.4.8 示例：重载

在下面的例子中：

```
class Point {
    final int EDGE = 20;
    int x, y;
    void move(int dx, int dy) {
        x += dx; y += dy;
        if (Math.abs(x) >= EDGE || Math.abs(y) >= EDGE)
            clear();
    }
    void clear() {
        System.out.println("\tPoint clear");
        x = 0; y = 0;
    }
}

class ColoredPoint extends Point {
    int color;
    void clear() {
        System.out.println("\tColoredPoint clear");
        super.clear();
        color = 0;
    }
}
```

子类 `ColoredPoint` 扩展了超类 `Point` 定义的 `clear` 抽象。它这样做是通过用自己的方法重载 `clear` 方法的，它使用形式 `super.clear` 调用了它的超类的 `clear` 方法。

然后在 `clear` 的调用的目标对象是一个 `ColoredPoint` 时，就会调用此方法。甚至

当 `this` 的类是 `ColoredPoint` 时, `Point` 中的方法 `move` 调用类 `ColoredPoint`, 如该测试程序的输出中所展示:

```
class Test {
    public static void main(String[] args) {
        Point p = new Point();
        System.out.println("p.move(20,20):");
        p.move(20, 20);
        ColoredPoint cp = new ColoredPoint();
        System.out.println("cp.move(20,20):");
        cp.move(20, 20);
        p = new ColoredPoint();
        System.out.println("p.move(20,20), p colored:");
        p.move(20, 20);
    }
}
```

其为:

```
p.move(20,20):
    Point clear
cp.move(20,20):
    ColoredPoint clear
    Point clear
p.move(20,20), p colored:
    ColoredPoint clear
    Point clear
```

重载有时称为“迟绑定自引用 (late-bound self-reference)”; 在本例中, 它意味着 `Point.move` 的体中对 `clear` 的引用 (这实际是 `this.clear` 句法速记。)调用了“迟”选择 (在运行时, 基于 `this` 引用的对象的运行时类) 的方法, 而不是“早”选择 (在编译时, 只基于 `this` 的类型) 的方法。这给程序员提供了一个扩展抽象的强大方式, 并且是面向对象程序中的一个关键方法。

#### 15.12.4.9 示例: 使用 `super` 的方法调用

通过使用关键字 `super` 来访问直接超类的成员, 跳过包含方法调用的类中的任何重载声明, 可以访问超类的一个重载实例方法。

当访问实例变量时, `super` 意味与 `this` 的转换相同 (15.11.2 节), 但这种等价对于方法调用是不成立的。这通过下面的例子来展示:

```
class T1 {
    String s() { return "1"; }
}
class T2 extends T1 {
    String s() { return "2"; }
}
class T3 extends T2 {
```

```

        String s() { return "3"; }
    void test() {
        System.out.println("s()=\t\t"+s());
        System.out.println("super.s()=\t"+super.s());
        System.out.print("(T2)this.s()=\t");
        System.out.println((T2)this.s());
        System.out.print("(T1)this.s()=\t");
        System.out.println((T1)this.s());
    }
}
class Test {
    public static void main(String[] args) {
        T3 t3 = new T3();
        t3.test();
    }
}

```

这将产生下面的输出：

```

s()=          3
super.s()=     2
((T2)this).s()= 3
((T1)this).s()= 3

```

类型 T1 和 T2 的强制转换没有改变被调用的方法，因为要被调用的实例方法是根据 this 引用的对象的运行时类进行选择的。强制转换没有改变对象的类；它只检查类是否与指定的类型相兼容。

## 15.13 数组访问表达式

数组访问表达式引用是数组元素的变量：

*ArrayAccess:*

*ExpressionName [ Expression ]*

*PrimaryNoNewArray [ Expression ]*

数组访问表达式包含两个子表达式：数组引用表达式（在左括号前面）和索引表达式（在括号中）。注意，数组引用表达式可能是一个名称或者不是数组创建表达式（15.10 节）的任何主表达式。

数组引用表达式的类型必须是一个数组类型（称它为  $T[]$ ，元素类型为  $T$  的一个数组），否则，会产生一个运行时错误。然后，数组访问表达式的类型是将捕获转换（5.1.10 节）应用到  $T$  的结果。

索引表达式经过一元数值的提升，提升后的类型必须是 `int`。

数组引用的结果是一个类型为  $T$  的变量，即索引表达式的值选择的数组中的变量。该结果变量（它是数组的一个元素）从未被当成 `final`，即使数组引用是从一个 `final` 变



量获得的。

### 15.13.1 数据访问的运行时计算

数组访问表达式是使用下面的过程进行计算的：

- 首先，数组引用表达式进行计算。如果此计算突然结束，那么数组访问就出于相同的原因突然结束，并且索引表达式没有被计算。
- 否则，索引表达式进行计算。如果此计算突然结束，那么数组访问出于相同的原因突然结束。
- 否则，如果数组引用表达式的值是 `null`，那么就抛出一个 `NullPointerException`。
- 否则，数组引用表达式的值的确引用一个数组。如果索引表达式的值少于 0，或大于或等于数组的长度，那么就抛出 `ArrayIndexOutOfBoundsException`。
- 否则，数组访问的结果就是类型为 `T` 的变量，在数组中，该变量由索引表达式的值进行选择（注意，该结果变量——数组的一个元素——从未被当成是 `final`，即使数组引用表达式是一个 `final` 变量）。

### 15.13.2 示例：数组访问求值顺序

在数组访问中，在括号中的表达式的任何部分被计算前，括号左边的表达式表现为完全计算。例如，在表达式 `a[(a=b)[3]]` 中，表达式 `a` 在表达式 `(a=b)[3]` 前是完全进行计算的；这意味着 `a` 的原始值是在表达式 `(a=b)[3]` 计算时被提取和记住的。`a` 的原始值引用的该数组然后成为一个下标值，该值是另一个数组（可能是相同的数组）的第 3 个元素，该数组被 `b` 引用，并且现在也由 `a` 进行引用。

因此，示例：

```
class Test {
    public static void main(String[] args) {
        int[] a = { 11, 12, 13, 14 };
        int[] b = { 0, 1, 2, 3 };
        System.out.println(a[(a=b)[3]]);
    }
}
```

输出：

14

因为表达式的值等于 `a[b[3]]` 或 `a[3]` 或 14。

如果括号左边的表达式的计算突然结束，那么括号中就没有任何一部分将表现为已经计算。因此，下面的例子：

```
class Test {
    public static void main(String[] args) {
        int index = 1;
        try {
```

```

        skedaddle()[index=2]++;
    } catch (Exception e) {
        System.out.println(e + ", index=" + index);
    }
}
static int[] skedaddle() throws Exception {
    throw new Exception("Ciao");
}
}

```

输出:

```
java.lang.Exception: Ciao, index=1
```

因此嵌入的将 2 赋值给 index 从未发生。

如果数组引用表达式产生 null 而不是对数组的一个引用, 那么在运行时就抛出 `NullPointerException`, 但只有在所有数组访问表达式的部分已经计算之后, 并且只有这些计算正常完成时。从而, 如下例子:

```

class Test {
    public static void main(String[] args) {
        int index = 1;
        try {
            nada()[index=2]++;
        } catch (Exception e) {
            System.out.println(e + ", index=" + index);
        }
    }
    static int[] nada() { return null; }
}

```

输出:

```
java.lang.NullPointerException, index=2
```

因为在检查到一个空指针前, 嵌入的将 2 赋值给 index 发生了。作为一个相关的例子, 如下程序:

```

class Test {
    public static void main(String[] args) {
        int[] a = null;
        try {
            int i = a[vamoose()];
            System.out.println(i);
        } catch (Exception e) {
            System.out.println(e);
        }
    }
    static int vamoose() throws Exception {
        throw new Exception("Twenty-three skidoo!");
    }
}

```

始终输出：

```
java.lang.Exception: Twenty-three skidoo!
```

`NullPointerException` 从未发生，因为在索引操作的任何一部分发生之前，索引表达式必须是完全计算的，并且它包括了检查，以查看左边的操作数的值是否为 `null`。

## 15.14 后缀表达式

后缀表达式包括后缀++和--运算符。而且，如第 15.8 节所讨论，名称没有被认为是主表达式，但按语法分别进行处理，以避免某些不确定性。它们就只可以在这里（在后缀表达式的某个优先级上）进行交换。

*PostfixExpression:*

*Primary*

*ExpressionName*

*PostIncrementExpression*

*PostDecrementExpression*

### 15.14.1 表达式名称

计算表达式名称的规则在第 6.5.6 节中给出了。

### 15.14.2 后缀增量运算符++

*PostIncrementExpression:*

*PostfixExpression ++*

跟有一个++运算符的后缀表达式是一个后缀增量表达式。后缀表达式的结果必须是一个可转换到（5.1.8 节）数值类型的类型的变量，否则，会发生编译时错误。后缀增量表达式的类型是该变量的类型。后缀增量表达式的结果不是一个变量，而是一个值。

在运行时，如果操作数的计算突然结束，那么后续增量表达式也会出于相同的原因突然结束，并且没有增量发生。否则，值 1 被添加到变量的值，并且和被存储回到变量中。在加之前，在值 1 和变量的值上执行加、二元数值提升（5.6.2 节）。如有必要，和通过限制基本转换（5.1.3 节）进行限制，和/或服从于装箱转换（5.1.7 节）到它被存储之前的变量的类型。后缀增量表达式的值是新值被存储前变量的值。

注意，上面提及的二元数值提升可能包括拆箱转换（5.1.8 节）和值集合转换（5.1.13 节）。如有必要，值集合转换在将和存储在变量中前被应用到它当中。

声明 `final` 的变量不能被增量，除非它确定是无符号（16 章）空白最终变量（4.12.4 节），因为当将对这样的一个 `final` 变量的访问用作一个表达式时，结果是一个值，而不是一个变量。因此，它不能用作后缀增量运算符的一个操作数。

### 15.14.3 后缀减量运算符--

*PostDecrementExpression:*

*PostfixExpression --*

跟有一个--运算符的后缀表达式是一个后缀减量表达式。后缀表达式的结果必须是可转换到(5.1.8 节)数值类型的类型的变量,否则,会发生一个编译时错误。后缀减量表达式的类型是变量的类型。后缀减量表达式的结果不是一个变量,而是一个值。

在运行时,如果操作数表达式的计算突然结束,那么后缀减量表达式会出于相同的原因突然结束,并且没有减量发生。否则,值 1 从变量中减去,并且差额被存储回到变量中。在减之前,二元数值提升(5.6.2 节)在值 1 和变量的值上执行二元提升操作。如有必要,差额通过限制基本转换进行限制(5.1.3 节),和/或服从于在变量被存储前,装箱转换到变量的类型(5.1.7 节)。后缀减量表达式的值是在新值被存储之前变量的值。

注意,上面提及的二元数值提升可能包括拆箱转换(5.1.8 节)和值集合转换(5.1.13 节)。如有必要,在将差额存储在变量前,值集合转换被应用到差额。

声明为 final 的变量不能被减量,除非它明确是一个无符号(第 16 章)空白最终变量(4.12.4 节),因为当将对这样的 final 变量的访问用作表达式时,结果是一个值,而不是一个变量。从而,它不能用作后缀减量运算符的操作数。

## 15.15 一元运算符

一元运算符包括+、-、++、--、~、!,及转换运算符。具有一元运算符的表达式从右到左成组,因此--x 意味着与-(~x)相同:

*UnaryExpression:*

*PreIncrementExpression*

*PreDecrementExpression*

*+ UnaryExpression*

*- UnaryExpression*

*UnaryExpressionNotPlusMinus*

*PreIncrementExpression:*

*++ UnaryExpression*

*PreDecrementExpression:*

*-- UnaryExpression*

*UnaryExpressionNotPlusMinus:*

*PostfixExpression*

*~ UnaryExpression*

*! UnaryExpression*

*CastExpression*

为方便起见，这里重复了 15.16 节的下面结果：

*CastExpression:*

*( PrimitiveType ) UnaryExpression*

*( ReferenceType ) UnaryExpressionNotPlusMinus*

### 15.15.1 前缀增量运算符++

前面有一个++运算符的一元表达式是前缀增量表达式。一元表达式的结果必须可转换（5.1.8 节）到数值类型的类型的变量，否则，会发生编译时错误。前缀增量表达式的类型是变量的类型。前缀增量表达式的结果不是一个变量，而是一个值。

在运行时，如果操作数表达式的求值突然结束，那么前缀增量表达式会出于相同的因为突然结束，并且没有增量发生。否则，值 1 被添加到变量的值中，且和被存储回到变量中。在添加之前，在值 1 和变量的值上执行二元数值提升（5.6.2 节）。如有必要，和通过限制基本转换（5.1.3 节）进行限制，和/或服从在将变量进行存储前装箱转换（5.1.7 节）到变量的类型。前缀增量表达式的值是新值被存储后变量的值。

注意，前面提及的二元数值提升可能包括拆箱转换（5.1.8 节）和值集合转换（5.1.13 节）。如果有必要，在和被存储在变量中前，就将值集合转换应用到和。

声明为 final 的变量不能够被减量（除非它明确是一个无符号（第 16 章）空白最终变量（4.12.4 节）），因为当将对这种的一种 final 变量访问用作一个表达式时，结果是一个值，而不是一个变量。从而，它不能用作前缀增量运算符的操作数。

### 15.15.2 后缀减量运算符--

他必须增加，但是我必须减少。

——《约翰福音》第 3 章第 30 节

前面有一个--运算符的一元表达式是前缀减量表达式。一元表达式的结果必须是类型可转换（5.1.8 节）到数值类型的类型的变量，否则，会发生编译时错误。前缀减量表达式的类型是变量的类型。前缀减量表达式的结果不是一个变量，而是一个值。

在运行时，如果操作数表达式的计算突然结束，那么前缀减量表达式会出于相同的原因突然结束，并且没有减量发生。否则，值 1 从变量的值中减去，并且差额存储回到变量中。在减之前，在值 1 和变量的值上执行地二元数值提升（5.6.2 节）。如有必要，差额通过限制基本转换（5.1.3 节）进行限制，和/或服从在将变量存储前装箱转换（5.1.7 节）到变量类型。前缀减量表达式的值是新值被存储后变量的值。

注意，前面提及的二元数值提升可能包括拆箱转换（5.1.8 节）和值集合转换（5.1.13 节）。如有必要，在将差额存储在变量前，会将格式转换应用到差额。

声明为 final 的变量不能被减量，除非它明确是一个无符号（第 16 章）空白最终变量（4.12.4 节），因为当将对这样的 final 变量的访问用作表达式时，结果是一个值，而

不是一个变量。从而，不能将它用作前缀增量运算符的操作数。

### 15.15.3 一元加运算符+

一元+运算符的操作数表达式的类型必须是可以转换（5.1.8 节）到基本数值类型的类型，否则就会发生一个编译时错误。一元数值提升是在操作数上执行的。一元加表达式的类型是操作数的被提升类型。一元加表达式的结果不是一个变量，而是一个值，即使操作数表达式的结果是一个变量。

在运行时，一元加表达式的值是操作数的提升值。

### 15.15.4 一元减运算符-

听到声音，看到该表达式的所有符号，是如此令人愉快。  
——格特鲁德·斯坦因，《柔软的纽扣》

一元-运算符的操作数表达式的类型必须是可转换（5.1.8 节）到基本数值类型的类型，否则，就发生编译时错误。一元数值提升是在操作数上执行的。一元减表达式的类型是操作数的被提升类型。

注意，一元数值提升在值集合转换（5.1.13 节）上执行。不管被提升的操作数的值提取自什么值集合，一元求反操作被执行，并且结果提取自相同的值集合。然后该结果用于进一步值集合转换。

在运行时，一元减表达式的值是操作数的被提升值的算术求反。

对于整数数值，求反与 0 减去整数值相同。Java 编程语言使用两个整数的 2 的补数表示，并且 2 的实数值的范围不是对称的，因为最大负 int 或 long 的求反导致了相同的最大负数。上溢在这种情形中发生了，但没有抛出异常。对所有整数值  $x$ ， $-x$  等于  $(\sim x) + 1$ 。

对于浮点值，求反和从 0 减不是相同的，因为如果  $x$  是  $+0.0$ ，那么  $0.0 - x$  是  $+0.0$ ，但  $-x$  是  $-0.0$ 。一元减只颠倒浮点数的符号。下面是感兴趣的特殊例子：

- 如果操作数是 NaN，那么结果是 NaN（回想 NaN 没有符号）。
- 如果操作数是无穷大，那么结果就是相反符号的无穷大。
- 如果操作数是 0，那么结果是相反符号的 0。

### 15.15.5 位补码运算符~

一元~运算符的操作数表达式的类型必须是一个可以转换（5.1.8 节）到一个基本整数类型的类型，否则就会发生编译时错误。一元数值提升是在操作数上执行的。一元位补码表达式的类型是操作数的提升类型。

在运行时，一元位补码表达式的值是操作数的提升值的位补码；注意，在所有的情形中， $\sim x$  等于  $(-x) - 1$ 。



### 15.15.6 逻辑组补码操作!

一元!运算符的操作数表达式的类型必须是 `boolean` 或 `Boolean`, 否则就会发生一个编译时错误。一元逻辑补码表达式的类型是 `boolean`。

在运行时, 如有必要, 操作数服从于拆箱转换 (5.1.8 节); 一元逻辑求补表达式的值是 `true`, 前提是 (可能转换的) 操作数值是 `false`, 或者一元逻辑求补表达式的值为 `false`, 前提是 (可能被转换的) 操作数值是 `true`。

## 15.16 强制转换表达式

我在死人中间的日子已经过去了;  
在我周围, 我看到了,  
这些漫不经心的眼神会投向哪里,  
往昔强大的精神……

——Robert Southey (1774~1843), 《Occasional Pieces》, xviii

强制转换表达式在运行时将数值类型的值转换到另一个数值类型的类似值; 或者在编译时确认表达式的类型是 `boolean`; 或者在运行时检查引用值是否引用某个对象, 该对象与指定的引用类型相兼容。

*CastExpression:*

( *PrimitiveType Dims<sub>opt</sub>* ) *UnaryExpression*  
( *ReferenceType* ) *UnaryExpressionNotPlusMinus*

有关 *UnaryExpression* 和 *UnaryExpressionNotPlusMinus* 的区别的讨论, 请参见第 15.15 节。

强制转换表达式的类型是将捕获转换 (5.1.10 节) 应用于其名称出现在括号中类型的结果 (括号和它们包含的类型有时称为强制转换运算符)。转换表达式的类型不是一个变量, 而是一个值, 即使操作数表达式的结果是一个变量。

转换运算符对类型 `float` 或类型 `double` 的值的值集合 (4.2.3 节) 的选择没有影响。因此, 在精确浮点的 (15.4 节) 表达式中强制转换到类型 `float`, 不一定导致它的值被转换到浮点值集合的一个元素, 在不是精确浮点数表达式中强制转换到类型 `double`, 不一定导致它的值被转换到双精度值集合。

如果操作数的编译时类型可能从未根据强制转换规则转换到强制转换运算符指定的类型, 这就是一个编译时错误。否则, 在运行时, 操作数值通过强制转换 (如果有必要) 到强制转换运算符指定的类型。

在编译时, 一些强制转换导致了一个错误。在编译时, 一些强制转换可能被证明在编译时始终是正确的。例如, 它始终是正确的, 以便将类类型的值转换到它的超类的类型:

这样的一种强制转换应该在运行时不要求特殊的操作。最后，一些转换不可能被证明在编译时始终是正确的，或始终是不正确的。这样的一些强制转换在运行时要求一个测试。参阅第 5.5 节，以获取进一步信息。

如果不可在运行时发现强制转换，就抛出 `ClassCastException`。

## 15.17 乘法运算符

运算符 `*`、`/` 和 `%` 被称为乘法运算符。它们有相同的优先级，并且在句法上是左结合的（它们从左到中进行分组）。

*MultiplicativeExpression:*

*UnaryExpression*

*MultiplicativeExpression* \* *UnaryExpression*

*MultiplicativeExpression* / *UnaryExpression*

*MultiplicativeExpression* % *UnaryExpression*

乘法运算符的每个操作数的类型必须是一个可以转换（5.1.8 节）到基本类型的类型，否则，编译错误就发生了。二元数值提升在操作数上执行（5.6.2 节）。乘法表达式的类型是其操作数的提升类型。如果此被提升的类型是 `int` 或 `long`，那就执行整型算术；如果被提升的类型是 `float` 或 `double`，那么就执行浮点算术。

注意，二元数值提升执行拆箱转换（5.1.8 节）和值集合转换（5.1.13 节）。

### 15.17.1 乘运算符\*

万事万物应该尽量简单，而不是更简单。

——William of Occam (c.1320)

二元 `*` 运算符执行乘，产生它的操作数的乘积。如果操作数表达式没有副作用，那么乘就是一个可交换操作。尽管当操作数的所有类型是相同的时候整数乘是结合的，但浮点乘不是结合的。

如果整数乘溢出，那么结果就是算术乘积的低序位，该乘积以足够大的 2 的补码格式进行表示。结果，如果溢出发生，则结果的符号可能与两个操作数的算法乘积的符号不同。

浮点乘的结果是由 IEEE754 算术进行管理的。

- 如果任何一个操作数是 NaN，那么结果是 NaN。
- 如果结果不是 NaN，那么如果两个操作数有相同的符号，则结果的符号就是正的，如果操作数有不同的符号，则结果的符号是负的。
- 无限大乘 0 产生 NaN。
- 无限大乘无限大值产生一个有符号的无限大数。符号是由上面指出的规则确定的。

- 在其余的情形中，其中无限大或 NaN 都不涉及，计算了准确的算术乘积。然后选择浮点值集合：
  - ◆ 若乘表达式是 FP 严格的（15.4 节）：
    - ◇ 如果乘表达式的类型是 float，那么必须选择浮点值集合必须。
    - ◇ 如果乘表达式的类型是 double，那么必须选择 double 集合集合。
  - ◆ 若乘表达式不是 FP 严格的：
    - ◇ 如果乘表达式的类型是 float，那么在实现中可以选择浮点值集合或浮点扩展指数值集合。
    - ◇ 乘表达式的类型是 double，那么在实现中可以选择双精度值集合或双精度扩展指数值集合。

接下来，必须从选择的值集合中选择一个值以表示乘积。如果乘积的量级太大了以至于不能表示，我们就说操作溢出；然后结果是一个恰当符号的无穷大数。否则，乘积舍入了选择的值中的最近值，所使用的是 IEEE754 舍入和最近模式。Java 编程语言要求支持逐步的下溢，如 IEEE 754（4.2.4 节）所定义。

不管上溢、下溢或信息丢失可能出现的情形，乘运算符\*的求值从未抛出运行时异常。

## 15.17.2 除运算符/

高卢全境分为三部分。

——Julius Caesar, 《Commentaries on the Gallic Wars》（公元前 58 年）

二元/运算符执行除运算，产生它的操作数的商。左边的操作数是被除数，右边的操作数是除数。

整除舍入到 0。也就是，操作数  $n$  和  $d$  产生的商是一个整数值  $q$ ，而操作数  $n$  和  $d$  在二元数值提升（5.6.2 节）后是整数， $q$  的量级是尽可能地大，同时满足  $|d \cdot q| \leq |n|$ ；而且，当  $|n| \geq |d|$  时， $q$  是正的，并且  $n$  和  $d$  有相反的符号。有种特例没有满足这一规则：如果被除数是这种类型的最大可能量级的负整数，并且除数是 -1，那么整数溢出就发生，并且结果等于被除数。尽管溢出，但在这种情形中没有异常抛出。另一方面，如果整数除中除数的值是 0，那么就抛出 ArithmeticException。

浮点除的结果是由 IEEE 算术的规范确定的：

- 如果任何一个操作是 NaN，那么结果是 NaN。
- 如果结果不是 NaN，那么如果两个操作数具有相同的符号，则结果的符号是正的，如果操作数具有不同的符号，则结果的符号就是负的。
- 无限大除以无限大导致 NaN。
- 无限大除以有限大的值导致有符号无限大。符号是由上面的规则确定的。
- 有限大值除以无限大导致有符号 0。符号是由上面指出的规则确定的。
- 0 除以 0 导致 NaN；0 由任何其他有限值除导致一个有符号 0。符号由上面指出的规

则确定。

- 非 0 有限值除以 0 导致一个有符号无限大。符号由上面指出的规则确定。
- 在其余的都涉及一个无限大或 NaN 的情形中，计算准确的算术商。然后选择浮点值集合值。
  - ◆ 若除表达式是精确浮点的（15.4 节）：
    - ◇ 如果除表达式的类型是 float，那么必须选择浮点值集合。
    - ◇ 如果除表达式的类型是 double，那么必须选择双精度值集合。
  - ◆ 若除表达式不是精确浮点的：
    - ◇ 如果除表达式的类型是 float，那么在实现中可以选择 float 值集合或 float 扩展指数值集合。
    - ◇ 如果除表达式的类型是 double，那么双精度值集合或双精度扩展指数值集合在实现中可以被选择。

接下来，必须从选定的值集合中选择一个值来表示商。如果商的量级太大，不能进行表示，我们就说操作溢出；然后结果是一个有恰当符号的无限大数。否则，商使用 IEEE 754 舍入到最近模式来舍入到选择值集合中的最近值。Java 编程语言要求支持如 IEEE754（4.2.4 节）所定义的逐渐下溢。

不管上溢、下溢、被零除或丢失信息可能发生的事实，浮点除运算符/的计算从未抛出运行时异常。

### 15.17.3 余数运算符%

在那石座上还有这样的铭记：  
“我是奥西曼德斯，众王之王：  
那些所谓强悍者呵，谁能和我的业绩相比！”  
——雪莱，《奥西曼德斯》（1817）

二元%运算符据说从隐含的除法中产生它的操作数的余数；左边的操作数是被除数，右边的操作数是除数。

在 C 和 C++ 中，余数运算符只接受整数运算符，但在 Java 编程语言中，它也接受浮点操作数。

在二元数值提升（5.6.2 节）后是整数的操作数的余数操作产生一个结果值，使得  $(a/b)*b + (a\%b)$  等于  $a$ 。此标识保留，即使在下面特殊的环境中：被除数是它的类型的最大可能量级的负整数，并且除数是 -1（余数是 0）。从此规则可以得出余数操作的结果可以是负的，只要它的被除数是负的；并且可以是正的，只要被除数是正的；而且，结果的量级始终小于除数的量级。如果整数余数运算符的除数值是 0，就抛出 ArithmeticException。如下面例子所示：

5%3 产生 2                      （注意 5/3 产生 1）  
5%(-3) 产生 2                  （注意 5/(-3) 产生 -1）

$(-5)\%3$  产生  $-2$                       (注意  $(-5)/3$  产生  $-1$ )  
 $(-5)\%(-3)$  产生  $-2$                       (注意  $(-5)/(-3)$  产生  $1$ )

通过 $\%$ 运算符计算的浮点余数操作的结果与 IEEE 754 定义的余数操作所产生的结果是不同的。IEEE 754 余数操作计算来自舍入除法,而不是一个截除法的余数,因此,它的行为不类似于一般的整数余数运算符的行为。相反,Java 编程语言在浮点操作上定义了 $\%$ ,以便以类似于整数余数运算符的方式进行表现;这可以与 C 库函数 `fmod` 进行比较。IEEE 754 的余数操作可以通过库例程 `Math.IEEEremainder` 进行计算。

浮点余数操作的结果是由 IEEE 算术确定的:

- 如果操作数是 NaN,那么结果是 NaN。
- 如果结果不是 NaN,那么结果的符号等于被除数的符号。
- 如果被除数是无限大的,或除数是 0,或两者,那么结果是 NaN。
- 如果被除数是有限大的,并且除数是无限大的,那么结果等于被除数。
- 如果被除数是 0,并且除数是有限大的,那么结果等于被除数。
- 在其余的情形中,不涉及无限大、0 或 NaN,那么从被除数  $n$  除以除数  $d$  得到的浮点余数  $r$  是通过算术关系  $r=n-(d \cdot q)$  定义的,其中  $q$  是一个整数,只有在  $n/d$  是负数的情形下,该整数才是负的,只有在  $n/d$  是正的情形下,该整数才是正的,并且其量级尽可能地大,而不会超过真正的  $n$  和  $d$  的算术商的量级。

浮点余数运算符 $\%$ 的求值从未抛出运行时异常,即使右边的操作数是 0。上溢、下溢或精度丢失不可能发生:

如下示例:

$5.0\%3.0$  产生  $2.0$   
 $5.0\%(-3.0)$  产生  $2.0$   
 $(-5.0)\%3.0$  产生  $-2.0$   
 $(-5.0)\%(-3.0)$  产生  $-2.0$

## 15.18 加运算符

运算符 $+$ 和 $-$ 称为加运算符。它们有相同的过程,并且在句法上是左结合的(它们从左到右进行分组)。

*AdditiveExpression:*

*MultiplicativeExpression*

*AdditiveExpression* + *MultiplicativeExpression*

*AdditiveExpression* - *MultiplicativeExpression*

如果一个 $+$ 运算符的任何一个操作数的类型是 `String`,那么操作就是字符串串接。

否则, $+$ 运算符的每个操作数的类型必须是可转换(5.1.8 节)到基本数值类型的类型,否则,会出现编译时错误。

在每种情形中,二元 $-$ 运算符的每个操作数的类型必须是可以转换(5.1.8 节)到基本数值类型的类型,否则,会出现编译时错误。



### 15.18.1 字符串串接运算符+

伊甸园中一段不幸的事件后，增加了第五根线……  
—John Philip Sousa, 《The Fifth String》(1902)

如果只有一个操作数表达式的类型是 `String`，那么字符串转换就在另一个操作数上执行，以便在运行时产生字符串。结果是 `String` 对象 [ 新创建，除非表达式是一个编译时常量表达式 (15.28 节) ] 的一个引用，该对象是两个操作数字符串的串接。在新创新的字符串中，左边操作数的字符在右边的操作数的字符前面。如果类型为 `String` 的操作数是 `null`，那么就使用字符串 `"null"` 而不是那个操作数。

#### 15.18.1.1 字符串转换

任何类型可能通过字符串转换来转换成类型 `String`。

基本类型 `T` 的值 `x` 首先被转换到一个引用值，就像通过让它作为恰当类实例创建表达式的一个参数一样：

- 如果 `T` 是 `boolean`，那么使用 `new Boolean(x)`。
- 如果 `T` 是 `char`，那么使用 `new Character(x)`。
- 如果 `T` 是 `byte`、`short` 或 `int`，那么使用 `new Integer(x)`。
- 如果 `T` 是 `long`，那么使用 `new Long(x)`。
- 如果 `T` 是 `float`，那么使用 `new Float(x)`。
- 如果 `T` 是 `double`，那么使用 `new Double(x)`。

然后将此引用值通过字符串转换转换到类型 `String`。

现在只有引用值需要考虑。如果引用是 `null`，它就被转换到字符串 `"null"` (4 个 ASCII 字符 `n`、`u`、`l`、`l`)。否则，转换就像通过不用参数调用被引用对象的 `toString` 方法一样执行；但如果调用 `toString` 方法的结果是 `null`，那么会使用字符串 `"null"` 进行替代。

`toString` 方法是通过原始类对象定义的；许多类重写它，特别地是 `Boolean`、`Character`、`Integer`、`Long`、`Float`、`Double` 和 `String`。

#### 15.18.1.2 字符串串接优化

实现可以选择在一个步骤中执行转换和串接，以避免创建及然后丢弃中间的 `String` 对象。要增加重复的字符串串接的性能，Java 编译器可以使用 `StringBuffer` 类或一种类似的技术来减少中间 `String` 对象的数量，这些 `String` 对象是通过求表达式的值创建的。

对于基本类型，实现也可以通过直接从基本类型转换到字符串来优化包装对象的创建方式。

#### 15.18.1.3 字符串串接的示例

示例表达式：

```
"The square root of 2 is " + Math.sqrt(2)
```

产生如下结果：



\*The square root of 2 is 1.4142135623730952\*

+运算符在句法上是左结合的，而不管它是否是在后面由类型分析确定的，以表示字符串串接或添加。在某些情形下，需要注意以获得所需的结果。例如，下面表达式：

`a + b + c`

始终被认为表达下面的含义：

`(a + b) + c`

因此表达式的结果：

`1 + 2 + " fiddlers"`

是：

`"3 fiddlers"`

但下面的结果：

`"fiddlers" + 1 + 2`

是：

`"fiddlers 12"`

在下面这个有意思的小例子中：

```
class Bottles {
    static void printSong(Object stuff, int n) {
        String plural = (n == 1) ? "" : "s";
        loop: while (true) {
            System.out.println(n + " bottle" + plural
                + " of " + stuff + " on the wall,");
            System.out.println(n + " bottle" + plural
                + " of " + stuff + ";");
            System.out.println("You take one down "
                + "and pass it around:");
            --n;
            plural = (n == 1) ? "" : "s";
            if (n == 0)
                break loop;
            System.out.println(n + " bottle" + plural
                + " of " + stuff + " on the wall!");
            System.out.println();
        }
        System.out.println("No bottles of " +
            stuff + " on the wall!");
    }
}
```

方法 `printSong` 将输出孩子的一个版本的歌曲。素材的通俗值包括 "pop" 和 "beer"；`n` 的最通俗的值是 100。下面是从 `Bottles.printSong("slime", 3)` 产生的输出：

`3 bottles of slime on the wall,`

```
3 bottles of slime;  
You take one down and pass it around:  
2 bottles of slime on the wall!
```

```
2 bottles of slime on the wall,  
2 bottles of slime;  
You take one down and pass it around:  
1 bottle of slime on the wall!
```

```
1 bottle of slime on the wall,  
1 bottle of slime;  
You take one down and pass it around:  
No bottles of slime on the wall!
```

在此代码中，在适当时注意单数“bottle”的仔细条件生成，而不是复数“bottles”；也注意字符串串接运算符如何用于将下面的长的常量字符串分开：

```
* You take one down and pass it around:*
```

分成了两块，以避免在源代码中出现不方便的长行。

### 15.18.2 数值类型的加运算符（+和-）

我们知道，恋爱中的人充满激情，单身者则缺乏这种激情；  
但是单身者有着清醒的头脑，而恋爱中的人则往往头脑发热。  
越偏见就越狭隘，爱情虽然带来了激情，同时也缩小了胸襟。

——托马斯·哈代，《远离尘嚣》（1874）第四幕第一场

当将二元+运算符加应用于数值类型的两个操作数时，它执行加，并产生操作数的和。二元-运算符执行减时，产生两个数值操作数的差额。

二元数值提升在操作数上执行（5.6.2 节）。数值操作数上的回表达式的类型是它的操作数的提升类型。如果此提升类型是 float 或 double，那么就执行浮点算术。

注意，二元数值提升执行值集合转换（5.1.3 节）和拆箱转换（5.1.8 节）。

如果操作数表达式没有副作用，那么加是一个可交换的操作。当操作数都是相同的类型时，整数加是结合的，但浮点加不是结合的。

如果整数加上溢，那么结果是算术和的低序位按足够大的 2 的补码格式进行表示。如果溢出发生，那么结果的符号就与两个操作数值的算术和一样。

浮点和的结果是使用下面的 IEEE 算术规则确定的：

- 如果任何一个操作数是 NaN，那么结果是 NaN。
- 两个符号相反的无穷数的和是 NaN。
- 两个符号相同的无穷数的和是那个符号的无穷数。
- 无穷数和有穷数数值的和等于无穷的操作数。
- 方向相反的两个 0 的和是正 0。
- 相同符号的两个 0 的和是那个符号的 0。

- 0 和非 0 有限值的和等于非 0 操作数。
- 相同量级且符号相反的两个非 0 有限值是正 0。
- 在其余的情形中，不涉及无穷大、0、NaN，那么操作数就有相同的符号或有不同的量级，计算准确的算术和。然后选择浮点值集合：
  - ◆ 若加表达式是精确浮点的（15.4 节）：
    - ◇ 如果加表达式的类型是 `float`，那么必须选择浮点值集合。
    - ◇ 如果加表达式的类型是 `double`，那么必须选择双精度值集合。
  - ◆ 若加表达式不是精确浮点的：
    - ◇ 如果加表达式的类型是 `float`，则要么是浮点值集合要么是浮点扩展指数值集合可能在实现中被选择。
    - ◇ 如果加表达式的类型是 `double`，则要么是双精度值集合要么是双精度扩展指数值集合可能在实现中被选择。

接下来，必须从选定的值集合中选择一个值来表示和。如果和的量级太大以至于不能表示，我们就说操作溢出；然后结果是一个适当符号的无穷大数。否则，和使用 IEEE 754 舍入到最近模式被舍入到被选定的值集合中的最近数值。Java 编程语言要求支持如 IEEE 754（4.2.4 节）所定义的逐渐下溢。

当将二元-运算符应用到两个数值类型的操作数时，它执行减操作，产生它的操作数的差额：左边的操作数是被减数，右边的操作数是减数。对于整数或浮点减，始终是这样的情形： $a-b$  产生了与  $a+(-b)$  相同的结果。

注意，对于整数值，从 0 减与非是是一样的，但对于浮点操作数从 0 减与是不一样的，因为如果  $x$  是  $+0.0$ ，那么  $0.0-x$  就是  $+0.0$ ，但  $-x$  是  $-0.0$ 。

不管上溢、下溢或信息丢失可能发生的事实，数值加运算符的求值从未抛出一个运行时异常。

## 15.19 移位运算符

我变成了不幸的人，还能说什么呢？  
……除了改变你自己外，“你还能对别人说什么？”  
——托马斯·潘恩，《北美危机》（1780）

移位运算符包括左移位 `<<`、有符号右移位 `>>` 和无符号右移位 `>>>`；它们是句法上左结合的（它们从左到右成组）。移位运算符的左边运算符是要被移动的值；右边的操作数指定了移位距离。

*ShiftExpression:*

*AdditiveExpression*

*ShiftExpression* `<<` *AdditiveExpression*

*ShiftExpression* `>>` *AdditiveExpression*

*ShiftExpression* `>>>` *AdditiveExpression*

移位运算符的每个操作数的类型必须是一个可以转换(5.1.8 节)到基本整数类型的类型的, 否则, 会发生编译时错误。二元数值提升(5.6.2 节)不是在操作数上执行的; 相反, 一元数组提升是在每个操作数上分别执行的。移位表达式的类型是左边操作数的提升类型。

如果左边操作数的提升类型是 `int`, 那么只有右边操作数的 5 个低序位被用作移位距离。就像右边操作数服从于具有掩码值 `0x1f` 的位逻辑 AND 运算符 `&` (15.22.1 节)。因此, 实际使用的移位距离始终在范围 0~31 (包括 0 和 31)。

如果左边操作数的提升类型是 `long`, 那么只有右边操作数的 6 个低序位被用作移位距离。这就像右边操作数被服从于位逻辑 AND 运算符 `&` (15.22.1 节), 并具有掩码值 `0x3f`。因此实际使用的移位距离始终在范围 0~63 (包括 0 和 63)。

在运行时, 移位操作是在左边操作数值的 2 的补码整数表示上执行的。

$n \ll s$  的值是  $n$  左移  $s$  个位置; 这等价 (即使上溢发生) 剩以 2 的乘方  $s$ 。

$n \gg s$  的值是  $s$  右移  $s$  个位置, 并进行了符号扩展。结果值是  $[n/2^s]$ 。对于  $n$  的非负值, 这等价于截整数除数, 如由整数除法运算符 `/` 所计算, 除以 2 的平方。

$n \ggg s$  的值是  $s$  被右移  $s$  个位置的, 没有进行 0 扩展。如果  $n$  是正的, 那么结果就与  $n \gg s$  的相同的; 如果  $n$  是负的, 那么结果就等于表达式  $(n \gg s) + (2 \ll \sim s)$  的结果, 前提是左边的操作数是 `int`, 并且等于表达式  $(n \gg s) + (2L \ll \sim s)$  的结果, 前提是右边的操作数的类型是 `long`。被添加的项  $(2 \ll \sim s)$  或  $(2L \ll \sim s)$  抵偿了被传播符号位。注意, 因为移动运算符的右边操作数的隐式掩码, 所以作为移位距离的  $\sim s$  等价于  $31-s$  (当移动 `int` 值时), 及等价于  $63-s$  (当移动 `long` 值时。)

## 15.20 关系运算符

关系运算符在句法上是右结合的 (它们从左到右成组), 但该事实是没有用的; 例如,  $a < b < c$  解析为  $(a < b) < c$ , 这始终是一个编译时错误, 因为  $a < b$  的类型始终是 `boolean`, 并且在 `<` 在 `boolean` 值上不是一个运算符。

*RelationalExpression:*

*ShiftExpression*

*RelationalExpression* `<` *ShiftExpression*

*RelationalExpression* `>` *ShiftExpression*

*RelationalExpression* `<=` *ShiftExpression*

*RelationalExpression* `>=` *ShiftExpression*

*RelationalExpression* `instanceof` *ReferenceType*

关系表达式的类型始终是 `boolean`。

### 15.20.1 数值比较运算符 `<`、`<=`、`>` 和 `>=`

数值比较运算符的每个操作数的类型必须是一个可以转换(5.1.8 节)到基本类型的类型, 否则会发生编译时错误。二元数值提升在这些操作数上执行(5.6.2 节)。如果操作数

的提升类型是 `int` 或 `long`, 那么就执行有符号整数比较; 如提升类型是 `float` 或 `double`, 那么就执行浮点比较。

注意, 二元提升执行值集合转换 (5.1.13 节) 和拆箱转换 (5.1.8 节)。比较在浮点数值上准确地执行, 而不管其表示值来自于什么值集合。

如 IEEE 754 的规范所确定, 浮点数比较的结果是:

- 如果任何一个操作数是 NaN, 那么结果是 `false`。
- 除了 NaN 外的所有值都进行比较, 与小于所有有限值的负无穷大值及大于所有有限值的正无限大进行比较。
- 正 0 和负 0 被认为是相等的。因此,  $-0.0 < 0.0$  是 `false` (例如), 但  $-0.0 \leq 0.0$  是正确的 (但请注意, 方法 `Math.min` 和 `Math.max` 将负 0 当成严格上小于正 0 的)。

服从浮点数的这些考虑, 然后下面的规则对于除了 NaN 外的整数操作数或浮点操作数来说是成立的:

- `<` 运算符产生的值是 `true`, 前提是左边操作数的值小于右边操作数的值, 否则为 `false`。
- `<=` 运算符产生的值是 `true`, 前提是左边操作数的值小于或等于右边操作数的值, 否则为 `false`。
- `>` 运算符产生的值是 `true`, 前提是左边操作数的值大于右边操作数的值, 否则是 `false`。
- `>=` 运算符产生的值是 `true`, 前提是左操作数的值大于或等于右边操作数的值, 否则为 `false`。

## 15.20.2 类型比较运算符 `instanceof`

`instanceof` 运算符的 *RelationalExpression* 操作数的类型必须是一个引用类型或 `null` 类型; 否则, 会发生编译时错误。在 `instanceof` 运算符后面提及的 *ReferenceType* 必须表示一个引用类型; 否则, 会发生一个编译时错误。如果 `instanceof` 运算符后面提及的 *ReferenceType* 没有表示一个泛型具体化类型, 那么这就是一个编译时错误。

在运行时, 如果 *RelationalExpression* 不是 `null`, 并且引用可以强制转换 (15.16 节) 到 *ReferenceType*, 而不会产生 `ClassCastException`, 那么 `instanceof` 运算符的结果就是 `true`。否则结果为 `false`。

如果将 *RelationalExpression* 转换到 *ReferenceType* 会被拒绝为一个编译时错误, 那么 `instanceof` 关系表达式同样会发生编译时错误。在这样的情形中, `instanceof` 的结果不可能是 `true`。

考虑下面示例程序:

```
class Point { int x, y; }
class Element { int atomicNumber; }
class Test {
    public static void main(String[] args) {
```



```
Point p = new Point();
Element e = new Element();
if (e instanceof Point) { // compile-time error
    System.out.println("I get your point!");
    p = (Point)e; // compile-time error
}
}
```

这个示例导致了两个编译时错误。强制转换(Point)就是不正确的,因为没有Element的任何实例或任何它的可能子类(这里没有展示任何一个)可能是Point的任何子类的实例。出于相同的原因,instanceof表达式也是不正确的。另一方面,如果类Point是Element的子类(在本例中,一个公认的奇怪想法):

```
class Point extends Element { int x, y; }
```

那么转换会是可能的,尽管它会要求一个运行时检查;然后instanceof表达式会是敏感和有效的。转换(Point) e不会产生一个异常,因为如果e的值不被正确地强制转换到类型Point,它就可能不会被执行。

## 15.21 相等运算符

相等运算符在句法上左结合的(它从左到右进行地分组),但这种事实未必是有用的;例如, `a==b==c` 解析成 `(a==b)==c`。 `a==b` 的结果类型始终是 `boolean`, 而且 `c` 必须是类型 `boolean`, 否则会发生一个编译时错误。因此, `a==b==c` 并没有测试以查看 `a`、`b` 和 `c` 都是相等的。

*EqualityExpression:*

*RelationalExpression*

*EqualityExpression* == *RelationalExpression*

*EqualityExpression* != *RelationalExpression*

`==` (等于) 和 `!=` (不等于) 运算符类似于关系运算符, 只是它们的优先级更低。因此, 当 `a<b` 和 `c<d` 有相同的真值时, `a<b==c<d` 是 `true`。

相等运算符可用于比较两个可以转换(5.1.8节)到数值类型的操作数, 或者两个类型为 `boolean` 或 `Boolean` 的操作数, 或两个引用类型或空类型其中一个类型的操作数。所有其他情形会导致编译时错误。相等表达式的类型始终是 `boolean`。

在所有情形中, `a!=b` 产生与 `!(a==b)` 相同的结果。如果操作数表达式没有副作用, 那么相等操作就是可交换的。

### 15.21.1 数值相等运算符==和!=

如果相等运算符的操作数都是数值类型, 或者一个是数值类型, 另一个可转换(5.1.8节)到数值类型, 那么就可以在这些操作数上执行二元数值提升(5.6.2节)。如果操作数



提升类型是 `int` 或 `long`, 那么就执行整数相等性测试; 如果提升类型是 `float` 或 `double`, 那么就执行浮点相等性测试。

注意, 二元数值提升执行值集合转换 (5.1.13 节) 和拆箱转换 (5.1.8 节)。比较在浮点数值上精确地执行, 而不管其表示的值是从什么值集合而来的。

浮点相等性测试是根据 IEEE 754 标准规则执行的:

- 如果任何一个操作数是 NaN, 那么 `==` 的结果是 `false`, 但 `!=` 的结果是 `true`。的确, 当且仅当 `x` 的值是 NaN 时, 测试 `x!=x` 为真 (方法 `Float.isNaN` 和 `Double.isNaN` 也可用于测试值是否为 NaN)。
- 正 0 和负 0 被当成相等的。因此, `-0.0==0.0` 是 `true` (例如)。
- 否则, 两个不同的浮点值被相等运算符当成是不相等的。尤其是一个值表示正无穷大, 一个值表示负无穷大时; 每次比较时只能等于它本身 不能等于其他任何值。

遵从浮点数的这些考虑, 下面的规则对于整型操作数或对于除 NaN 外的浮点操作数是成立的:

- 如果左边操作数的值等于右边操作数的值, `==` 运算符产生的值是 `true`; 否则, 结果是 `false`。
- 如果左边操作数的值不等于右边操作数的值, `!=` 运算符产生的值就是 `true`。否则, 值为 `false`。

### 15.21.2 布尔相等运算符 `==` 和 `!=`

如果相等运算符的操作数两者的类型都是 `boolean`, 或者如果一个操作数的类型是 `boolean`, 另一个的类型是 `Boolean`, 那么操作就是布尔相等。布尔相等运算符是结合的。

如果其中一个操作数是 `Boolean`, 那么它就遵从拆箱转换 (5.1.8 节)。

如果两个操作数 (在任何所需的拆箱转换之后) 都是 `true` 或都是 `false`, 那么 `==` 的结果是 `true`, 否则, 结果是 `false`。

如果两个操作数都是 `true`, 或两者都是 `false`, 那么 `!=` 的结果是 `false`; 否则, 结果是 `true`。因而, 当将 `!=` 应用于布尔操作数时, 它的表现与 `^` (15.22.2 节) 一样。

### 15.21.3 引用相等运算符 `==` 和 `!=`

事情更可能是它们现在这样, 而不是过去那样。

——《Dwight D.Eisenhower》

如果相等运算符的两个操作数要么是引用类型要么是空类型, 那么运算就是对象相等性。

如果可以通过强制转换转换 (5.5 节) 将任一个操作数的类型转换到另一个操作数的类型, 那么就会发生编译时错误。两个操作数的运行时规则一定要不相等。

在运行时, 如果两个操作数值都是 `null` 或都引用相同的对象或数组, 那么 `==` 的结果就是 `true`; 否则, 结果是 `false`。

如果两个操作数值都是 `null` 或都引用相同的对象或数组, 那么 `!=` 的结果就是 `false`;

否则，结果是 true。

当==可用于比较类型为 String 的引用时，这样的相等测试确定两个操作数是否引用相同的 String 对象。如果操作数是不同的 String 对象，那么结果就是 false，即使它们包含相同的字符序列。两个字符串 s 和 t 的内容可以通过方法调用 s.equals(t) 对其相等性进行测试。也请参阅第 3.10.5 节。

## 15.22 位和逻辑运算符

位运算符和逻辑运算符包含 AND 运行符&，不包括 OR 运算符^，及包括 OR 运算符|。这些运算符有不同的优先级，&具有最高的优先级，|具有最低的优先级。这些运算符中的每一个在句法上是左结合的（每个从左到右分组）。如果操作数没有副作用，那么每个运算符是可交换的。

*AndExpression:*

*EqualityExpression*

*AndExpression & EqualityExpression*

*ExclusiveOrExpression:*

*AndExpression*

*ExclusiveOrExpression ^ AndExpression*

*InclusiveOrExpression:*

*ExclusiveOrExpression*

*InclusiveOrExpression | ExclusiveOrExpression*

位和逻辑运算符可用于比较数值类型的两个操作数或类型为 boolean 的两个操作数。所有其他情形导致编译时错误。

### 15.22.1 整数位运算符&、^和|

当运算符&、^或|的两个操作数的类型是可转换（5.1.8 节）到基本类型的类型时，二元数值提升首先在操作数上执行（5.6.2 节）。位运行符表达式的类型是操作数的提升类型。

对于&，结果值是操作数值的位与。

对于^，结果值是操作数值的位异域。

对于|，结果值是操作数的位同域（inclusive OR）。

例如，表达式 0xff00 & 0xf0f0 的结果是 0xf000。0xff00^0xf0f0 的结果是 0x0ff0。0xff00|0xf0f0 的结果是 0xffff0。

### 15.22.2 布尔逻辑运算符&、^和|

当&、^或|运算符的两个操作数的类型是 boolean 或 Boolean 时，那么位运算符表达式的类型就是 boolean。在所有的情形中，操作数根据需要遵从拆箱转换（5.1.8 节）。

对于&, 如果两个操作数值是 true, 那么结果值是 true; 否则, 结果是 false。  
对于^, 如果操作数值是不同的, 那么结果是 true; 否则, 结果是 false。  
对于!, 如果两个操作数值是 false, 那么结果值是 false; 否则, 结果是 true。

## 15.23 条件与运算符&&

&&运算符像& (15.22.2 节) 一样, 但只有在它的左边操作数的值是 true 时才求它的右边操作数值。从句法上讲, 它是左结合的(它从左到右进行组合)。至于副作用和结果值, 它是完全结合的; 也就是, 对任何表达式 *a*、*b* 和 *c*, 表达式  $((a) \&\& (b)) \&\& (c)$  的计算产生与表达式  $(a) \&\& ((b) \&\& (c))$  的计算相同的结果, 具有按相同顺序出现的副作用。

*ConditionalAndExpression:*

*InclusiveOrExpression*

*ConditionalAndExpression* && *InclusiveOrExpression*

&&的每个操作数的类型必须是 boolean 或 Boolean, 否则会发生编译时错误。条件与表达式的类型始终是 boolean。

在运行时, 左边操作数表达式是最先计算的; 如果结果的类型是 Boolean, 那么它就遵从拆箱转换 (5.1.8 节); 如结果值是 false, 那么条件与表达式的值就是 false, 并且右边操作数表达式没有进行计算。如果左边操作数的值是 true, 那么右边表达式就进行计算; 如果结果具有类型 Boolean, 那么它就遵从拆箱 (5.1.8 节); 结果值变成条件表达式的值。因而, 在布尔操作数上&&与&计算结果相同。它的不同只在于右边表达式是有条件而不是总是进行计算的。

## 15.24 条件或运算符||

||运算符像| (15.22.2 节), 但只有在它的左边操作数的值是 false 时才计算它的右边操作数。从句法上, 它是左结合的(它从左到右进行分组)。关于副作用和结果值, 它是完全结合的。也就是说, 对于任何表达式 *a*、*b* 和 *c*, 表达式  $((a) || (b)) || (c)$  的求值产生与表达式  $(a) || ((b) || (c))$  的求值相同的结果, 具有以相同顺序出现的副作用。

*ConditionalOrExpression:*

*ConditionalAndExpression*

*ConditionalOrExpression* || *ConditionalAndExpression*

||的每个操作数的类型必须是 boolean 或 Boolean, 否则, 会出现编译时错误。条件或表达式的类型始终是 boolean。

在运行时, 首先计算右边操作数表达式; 如果结果具有类型 Boolean, 那么就遵从拆箱转换 (5.1.8 节); 如果结果值是 true, 那么条件或表达式的值就是 true, 并且右边的操作数表达式没有进行计算。如果左边操作数的值是 false, 那么右边表达式就被计算; 如果结果具有类型 Boolean, 那么它就遵从拆箱转换 (5.1.8 节); 结果值变成条件或表达式的值。

因此，在 `boolean` 或 `Boolean` 操作数上，`||` 与 `|` 计算出相同的结果。它的惟一不同在于右边操作数表达式是有条件而不是总是进行计算的。

## 15.25 条件运算符?:

不论如何，我这里决定把王位永远让给你和你的子孙，  
但必须附一条件，那就是……  
——威廉·莎士比亚，《亨利六世》（1623）第三卷第一幕第一场

条件运算符?:使用一个表达式的 `boolean` 值来确定两个表达式中的哪个表达式应该进行计算。

从句法上，条件运算符是右结合的（它从右到左进行组合），因此 `a?b:c?d:e?f:g` 的意义是 `a?b:(c?d:(e?f:g))`。

*ConditionalExpression:*

*ConditionalOrExpression*

*ConditionalOrExpression ? Expression : ConditionalExpression*

条件运算符有 3 个操作数表达式：“?” 出现在第一个和第二个表达式之间，并且 “:” 出现在第二个和第三个表达式之间。

第一个表达式的类型必须是 `boolean` 或 `Boolean`，否则会出现编译时错误。

注意，让第二个或第三个操作数表达式调用一个 `void` 方法，会发生编译时错误。事实上，不允许条件表达式出现在调用 `void` 方法可能出现（14.8 节）的任何上下文中。

条件表达式的类型像下面那样进行确定：

- 如果第二个和第三个操作数具有相同的类型（该类型可能是一个空类型），那么它就是条件表达式的类型。
- 如果第二个或第三个操作数中的一个的类型是 `boolean`，并且另一个的类型是 `Boolean`，那么条件表达式的类型是 `boolean`。
- 如果第二个和第三个操作数中的一个为空类型，并且另一个的类型是引用类型，那么条件表达式的类型就是该引用类型。
- 否则，如果第二个和第三个操作数具有可转换（5.1.8 节）到数值类型的类型，那么就有这几种情形：
  - ◆ 若操作数之一的类型为 `byte` 或 `Byte`，另一个操作数的类型是 `short` 或 `Short`，那么条件表达式的类型是 `short`。
  - ◆ 若操作数之一是 `T`，其中 `T` 是 `byte`、`short` 或 `char`，并且其他操作数是类型 `int` 的常量表达式，其值可用类型 `T` 表示，那么条件表达式的类型是 `T`。
  - ◆ 若操作数之一的类型是 `Byte`，并且另一个操作数是类型为 `int` 的常量表达式，其值可用类型 `byte` 表示，那么条件表达式的类型是 `byte`。
  - ◆ 若操作数之一是类型 `Short`，并且另一个操作数的类型为 `int` 的常量表达式类

型，其值可用类型 `short` 表示，那么条件表达式的类型就是 `short`。

- ◆ 若操作数之一的类型是 `Character`，并且另一个操作数是类型为 `int` 的常量表达式，其值可用类型 `char` 表示，那么条件表达式的类型就是 `char`。
- ◆ 否则，二元数值提升（5.6.2 节）被应用到操作数类型，并且条件表达式的类型是第二个和第三个操作数的提升类型。注意，二元数值提升执行拆箱转换（5.1.8 节）和值集合转换（5.1.13 节）。

- 否则，第二个和第三个操作数的类型分别是  $S1$  和  $S2$ 。令  $T1$  是从装箱转换到  $S1$  的类型的结果，并令  $T2$  是从装箱转换到  $S2$  的类型的结果。条件表达式的类型就是将捕获转换（5.1.10 节）应用到  $\text{lub}(T1, T2)$ （15.12.2.7 节）的结果。

在运行时，条件表达式的第一个参数表达式首先进行计算；如有必要，在就结果上执行拆箱转换；然后结果 `boolean` 值用于选择第二个或第三个操作数表达式：

- 如果第一个参数的值是 `true`，那么第二个参数表达式就被选择。
- 如果第一个参数的值是 `false`，那么第三个参数表达式就被选择。

然后计算选择的操作数表达式，并且将结果值转换到如上面指定的规则确定的表达式的类型。此转换可能包括装箱（5.1.7 节）或拆箱转换。没有选择的操作数表达式没有针对条件表达式的该特定计算进行计算。

## 15.26 赋值运算符

有 12 种赋值运算符：它们都是句法上右结合的（它们从右到左进行组合）。因此， $a=b=c$  意味着  $a=(b=c)$ ，它将  $c$  的值赋值给  $b$ ，然后将  $b$  的值赋值给  $a$ 。

*AssignmentExpression:*

*ConditionalExpression*

*Assignment*

*Assignment:*

*LeftHandSide AssignmentOperator AssignmentExpression*

*LeftHandSide:*

*ExpressionName*

*FieldAccess*

*ArrayAccess*

*AssignmentOperator: one of*

`= *= /= %= += -= <<= >>= >>>= &= ^= |=`

赋值运算符的第一个操作数的结果必须是一个变量，否则就发生编译时错误。该操作数可能是一个命名变量，比如一个局部变量或当前对象或类的一个字段，或者它可能是一个计算的变量，如可以从字符访问（15.11 节）或数组访问（15.13 节）产生。赋值表达式的类型是捕获转换（5.1.10 节）后变量的类型。

在运行时，赋值表达式的结果是赋值发生后变量的值。赋值表达式的结果本身不是一



个变量。

声明为 `final` 的变量不能被赋予[除非它确定是无符号的(第 16 章)]空白 `final` 变量(4.12.4 节), 因为当将对这样的 `final` 变量的访问用作表达式时, 结果是一个值, 而不是一个变量, 因此, 它不能用作赋值运算符的第一个操作数。

### 15.26.1 简单赋值运算符=

如果右边操作数的类型不能通过赋值转换(5.2 节)转换到变量的类型, 那么就发生编译时错误:

在运行时, 表达式在下面 3 种方式之一进行计算:

- 如果右边的操作数表达式是字段访问表达式(15.11 节) `e.f`, 可能封闭在一对或多对括号中, 那么:
  - ◆ 首先, 计算表达式 `e`。如果 `e` 的计算突然结束, 那么赋值表达式也出于相同的原因突然结束。
  - ◆ 接下来, 计算右边的操作数。如果右边操作数的计算突然结束, 那么赋值表达式出于相同的因为结束。
  - ◆ 然后, 如果 `e.f` 表示的字段不是 `static`, 并且上面 `e` 的求值是 `null`, 那么就抛出 `NullPointerException`。
  - ◆ 否则, `e.f` 表示的变量像上面计算那样被赋予右边操作数的值。
- 如果左边操作数是一个数组访问表达式(15.13 节), 该表达式可能被封闭在一对或多对括号中, 那么:
  - ◆ 首先, 计算右边操作数数组访问表达式的数组引用子表达式。如果此计算突然结束, 那么赋值表达式也出于相同的原因突然结束; (右边操作数数组访问表达式的) 此子表达式和右边操作数没有进行计算, 并且没有赋值发生。
  - ◆ 否则, 计算右边操作数数组访问表达式的索引子表达式。如果此计算突然结束, 那么赋值表达式也出于相同的原因突然结束, 并且右边操作数没有计算, 且没有赋值发生。
  - ◆ 否则, 计算右边操作数。如果此计算突然结束, 那么赋值表达式也出于相同的原因突然结束, 且没有赋值发生。
  - ◆ 否则, 如果数组引用子表达式的值是 `null`, 那么就没有赋值发生, 并抛出 `NullPointerException`。
  - ◆ 否则, 数组引用子表达式的值的确引用数组。如果索引子表达式的值小于 0, 或大于或等于数组的长度, 那么就没有赋值发生, 并抛出 `ArrayIndexOutOfBoundsException`。
  - ◆ 否则, 索引子表达式的值用于选择由数组引用子表达式的值引用的数组的元素。该元素是一个变量; 将它的类型称为 `SC`。而且令 `TC` 是在编译时确定的赋值运算符的左边操作数的类型。
  - ◆ 如果 `TC` 是一个基本类型, 那么 `SC` 一定与 `TC` 相同。右边操作数的值被转换到选



择的数组元素的类型，并遵从于从值集合转换（5.1.13 节）到适当的标准值集合（而不是一个扩展指数值集合），并且转换的结果存储在数组元素中。

- ◆ 如果  $T$  是一个引用类型，那么  $SC$  可能与  $TC$  不同，相反，是一个扩展或实现了  $TC$  的类型。令  $RC$  是运行时右边操作数的值引用的对象的类。

编译器可能能够证明能够在编译时证明数组元素正好是  $TC$  类型（例如， $TC$  可能是一个 `final`）。但如果编译器不能在编译时证明数组元素正好是类型  $TC$ ，那么在运行时必须执行一个检查，以确保类  $RC$  是与数组元素的实际类型  $SC$  相兼容（5.2 节）的赋值。此检查类似于限制强制转换（5.5 节，15.16 节），除了如果检查失败，会抛出一个 `ArrayStoreException`，而不是一个 `ClassCastException`。因此：

- ◆ 如果类  $RC$  对类型  $SC$  不是可赋予的，那么就没有赋值发生，并且抛出一个 `ArrayStoreException`。

- ◆ 否则，右边操作数的引用值被存储到选定的数组元素中。

- 否则，需要 3 个步骤。

- ◆ 首先，计算右操作数以产生一个变量。如果此计算突然结束，那么赋值表达式就出于相同原因而突然结束：右操作数没有计算，并且没有赋值发生。
- ◆ 否则，计算右边的操作数。如果此计算突然结束，那么赋值表达式也出于相同的原因突然结束，并且没有赋值发生。

否则，右操作数的值被转换到左边变量的类型，并遵从值集合转变（5.1.13 节）到适当的标准值集合（不是一个扩展指数值集合），并将转换的结果存储在变量中。赋值到数组元素的规则由下面的示例程序展示：

```
class ArrayReferenceThrow extends RuntimeException { }
class IndexThrow extends RuntimeException { }
class RightHandSideThrow extends RuntimeException { }
class IllustrateSimpleArrayAssignment {
    static Object[] objects = { new Object(), new Object() };
    static Thread[] threads = { new Thread(), new Thread() };
    static Object[] arrayThrow() {
        throw new ArrayReferenceThrow();
    }
    static int indexThrow() { throw new IndexThrow(); }
    static Thread rightThrow() {
        throw new RightHandSideThrow();
    }
    static String name(Object q) {
        String sq = q.getClass().getName();
        int k = sq.lastIndexOf('.');
        return (k < 0) ? sq : sq.substring(k+1);
    }
    static void testFour(Object[] x, int j, Object y) {
        String sx = x == null ? "null" : name(x[0]) + "s";
        String sy = name(y);
        System.out.println();
        try {
            System.out.print(sx + "[throw]=throw => ");
        }
    }
}
```

```

        x[indexThrow()] = rightThrow();
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    try {
        ystem.out.print(sx + "[throw]=" + sy + " => ");
        x[indexThrow()] = y;
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    try {
        System.out.print(sx + "[" + j + "]=" + throw + " => ");
        x[j] = rightThrow();
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    try {
        System.out.print(sx + "[" + j + "]=" + sy + " => ");
        x[j] = y;
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
}

public static void main(String[] args) {
    try {
        System.out.print("throw[throw]=throw => ");
        arrayThrow()[indexThrow()] = rightThrow();
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    try {
        System.out.print("throw[throw]=Thread => ");
        arrayThrow()[indexThrow()] = new Thread();
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    try {
        System.out.print("throw[1]=throw => ");
        arrayThrow()[1] = rightThrow();
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    try {
        System.out.print("throw[1]=Thread => ");
        arrayThrow()[1] = new Thread();
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    testFour(null, 1, new StringBuffer());
    testFour(null, 1, new StringBuffer());
    testFour(null, 9, new Thread());
    testFour(null, 9, new Thread());
    testFour(objects, 1, new StringBuffer());
    testFour(objects, 1, new Thread());
    testFour(objects, 9, new StringBuffer());
    testFour(objects, 9, new Thread());
    testFour(threads, 1, new StringBuffer());
    testFour(threads, 1, new Thread());
    testFour(threads, 9, new StringBuffer());
    testFour(threads, 9, new Thread());
}

```

```

}
}

```

此程序输出:

```

throw[throw]=throw => ArrayReferenceThrow
throw[throw]=Thread => ArrayReferenceThrow
throw[1]=throw => ArrayReferenceThrow
throw[1]=Thread => ArrayReferenceThrow

null[throw]=throw => IndexThrow
null[throw]=StringBuffer => IndexThrow
null[1]=throw => RightHandSideThrow
null[1]=StringBuffer => NullPointerException

null[throw]=throw => IndexThrow
null[throw]=StringBuffer => IndexThrow
null[1]=throw => RightHandSideThrow
null[1]=StringBuffer => NullPointerException

null[throw]=throw => IndexThrow
null[throw]=Thread => IndexThrow
null[9]=throw => RightHandSideThrow
null[9]=Thread => NullPointerException

null[throw]=throw => IndexThrow
null[throw]=Thread => IndexThrow
null[9]=throw => RightHandSideThrow
null[9]=Thread => NullPointerException

Objects[throw]=throw => IndexThrow
Objects[throw]=StringBuffer => IndexThrow
Objects[1]=throw => RightHandSideThrow
Objects[1]=StringBuffer => Okay!

Objects[throw]=throw => IndexThrow
Objects[throw]=Thread => IndexThrow
Objects[1]=throw => RightHandSideThrow
Objects[1]=Thread => Okay!

Objects[throw]=throw => IndexThrow
Objects[throw]=StringBuffer => IndexThrow
Objects[9]=throw => RightHandSideThrow
Objects[9]=StringBuffer => ArrayIndexOutOfBoundsException

Objects[throw]=throw => IndexThrow
Objects[throw]=Thread => IndexThrow
Objects[9]=throw => RightHandSideThrow
Objects[9]=Thread => ArrayIndexOutOfBoundsException

Threads[throw]=throw => IndexThrow
Threads[throw]=StringBuffer => IndexThrow
Threads[1]=throw => RightHandSideThrow
Threads[1]=StringBuffer => ArrayStoreException

Threads[throw]=throw => IndexThrow
Threads[throw]=Thread => IndexThrow
Threads[1]=throw => RightHandSideThrow
Threads[1]=Thread => Okay!

```

```

Threads[throw]=throw => IndexThrow
Threads[throw]=StringBuffer => IndexThrow
Threads[9]=throw => RightHandSideThrow
Threads[9]=StringBuffer => ArrayIndexOutOfBoundsException

Threads[throw]=throw => IndexThrow
Threads[throw]=Thread => IndexThrow
Threads[9]=throw => RightHandSideThrow
Threads[9]=Thread => ArrayIndexOutOfBoundsException

```

其中最有趣的是来自后面的 1/13:

```
Threads[1]=StringBuffer => ArrayStoreException
```

它指出尝试将对 `StringBuffer` 的引用存储到其元素类型为 `Thread` 的数组抛出一个 `ArrayStoreException`。代码在编译时是类型正确的: 赋值左边的类型为 `Object`, 右边的类型为 `Object`。在运行时, 方法 `testFour` 的第一个实参是对“`Thread` 的数组”实例的一个引用, 并且第 3 个实参是对类 `StringBuffer` 的实例的一个引用。

### 15.26.2 组合赋值运算符

形如  $E1 \text{ op } = E2$  的组合赋值表达式等价于  $E1 = (T)((E1) \text{ op } (E2))$ , 其中  $T$  的类型为  $E1$ , 除了  $E1$  只计算一次。

例如, 下面的代码是正确的:

```
short x = 3;
x += 4.6
```

并导致  $x$  有值 7, 因为它等价于:

```
short x = 3;
x = (short)(x + 4.6)
```

在运行时, 表达式用下面两种方式之一进行计算。如果左边操作数表达式不是一个数组访问表达式, 那么需要 4 个步骤:

- 首先, 计算右边操作数以产生一个变量。如果此计算突然结束, 那么赋值表达式就出于相同的原因突然结束; 右边操作数没有进行计算, 并且没有赋值发生。
- 否则, 保存右边操作数的值, 然后计算右边操作数。如果此计算突然结束, 那么赋值表达式就出于相同的原因突然结束, 并且没有赋值发生。
- 否则, 右边变量的保存值和右边操作数的值用于执行由组合赋值运算符指出的二元操作。如果此操作突然结束, 那么赋值表达式也出于相同的原因突然结束, 并没有赋值发生。
- 否则, 二元操作的结果被转换到右边值的类型, 遵从值集合转换 (5.1.13 节) 到适当的标准值集合 (不是一个扩展指数值集合)。转换的结果存储在该变量中。

如果右边操作数表达式是数组访问表达式 (15.13 节), 那么可能需要许多步骤:

- 首先, 计算左边操作数数组访问表达式的数组引用子表达式。如果此计算突然结束, 那么赋值表达式就出于相同的原因突然结束; (左边操作数数组访问表达式的) 索引

子表达式和右边的操作数没有进行计算，并且没有赋值发生。

- 否则，左边操作数数组访问表达式的索引子表达式进行计算。如果此计算突然结束，那么赋值表达式出于相同的原因突然结束，并且没有计算右边的操作数，且没有赋值发生。
- 否则，如果数组引用子表达式的值是 null，那么就没有赋值发生，并抛出一个 `NullPointerException`。
- 否则，数组引用子表达式的值的确引用一个数组。如果索引子表达式的值小于 0，或大于或等于数组的长度，那么就没有赋值发生，并抛出 `ArrayIndexOutOfBoundsException`。
- 否则，索引子表达式的值用于选择由数组引用子表达式的值引用的数组的一个元素。此元素的值被保存，然后右边操作数进行计算。如果此计算突然结束，那么赋值表达式也会出于相同的原因突然结束，并且没有赋值发生（对于简单的赋值运算符，右边操作数的求值在检查数组引用子表达式和索引子表达式之前发生，但对于组合赋值运算符，右边操作数的求值在这些检查之后发生）。
- 否则，考虑前一步骤中选择的数组元素，其值被保存了。此元素是一个变量；称它的类型为 *s*。而且，令 *T* 是在编译时确定的赋值运算符的左边操作数的类型。
  - ◆ 如果 *T* 是一个基本类型，那么 *s* 一定与 *T* 相同。
    - ◇ 数组元素的保存值和右边操作数的保存值用于执行由元素赋值运算符指出的二元操作。如果此操作数突然结束（惟一的可能性 0 除一个整数——15.17.2 节），那么赋值表达式会出于相同的原因突然结束。
    - ◇ 否则，二元操作的结果被转换到选定的数组元素的类型，遵从值集合转换（5.1.13 节）到适当的标准值集合（而不是扩展指数值集合），并且转换的结果被存储到数组元素。
  - ◆ 如果 *T* 是一个引用类型，那么它必须是 `String`。因为类 `String` 是一个 `final` 类，*s* 也必须是 `String`。因此有时对于简单赋值运算符需要的运行时检查对于一个组合赋值运算符来说从未是需要的。
    - ◇ 数组元素保存的值和右操作数的值用于执行组合赋值运算符（它一定是 +=）指出的二元操作（字符串串接）。如果此操作突然结束，那么赋值表达式会出于相同的原因突然结束，并没有赋值发生。

否则，二元操作的 `String` 结果存在在数组结果中。

组合赋值到数组元素的规则由下面的示例程序展示：

```
class ArrayReferenceThrow extends RuntimeException { }
class IndexThrow extends RuntimeException { }
class RightHandSideThrow extends RuntimeException { }
class IllustrateCompoundArrayAssignment {
    static String[] strings = { "Simon", "Garfunkel" };
    static double[] doubles = { Math.E, Math.PI };
    static String[] stringsThrow() {
        throw new ArrayReferenceThrow();
    }
}
```

```

static double[] doublesThrow() {
    throw new ArrayReferenceThrow();
}
static int indexThrow() { throw new IndexThrow(); }
static String stringThrow() {
    throw new RightHandSideThrow();
}
static double doubleThrow() {
    throw new RightHandSideThrow();
}
static String name(Object q) {
    String sq = q.getClass().getName();
    int k = sq.lastIndexOf('.');
    return (k < 0) ? sq : sq.substring(k+1);
}
static void testEight(String[] x, double[] z, int j) {
    String sx = (x == null) ? "null" : "Strings";
    String sz = (z == null) ? "null" : "doubles";
    System.out.println();
    try {
        System.out.print(sx + "[throw]+=throw => ");
        x[indexThrow()] += stringThrow();
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    try {
        System.out.print(sz + "[throw]+=throw => ");
        z[indexThrow()] += doubleThrow();
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    try {
        System.out.print(sx + "[throw]+=\"heh\" => ");
        x[indexThrow()] += "heh";
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    try {
        System.out.print(sz + "[throw]+=12345 => ");
        z[indexThrow()] += 12345;
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    try {
        System.out.print(sx + "[" + j + "]+=throw => ");
        x[j] += stringThrow();
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    try {

```



```

        System.out.print(sz + "[" + j + "]+=throw => ");
        z[j] += doubleThrow();
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    try {
        System.out.print(sx + "[" + j + "]+=\"heh\" => ");
        x[j] += "heh";
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    try {
        System.out.print(sz + "[" + j + "]+=12345 => ");
        z[j] += 12345;
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
}

public static void main(String[] args) {
    try {
        System.out.print(*throw[throw]+=throw => ");
        stringsThrow()[indexThrow()] += stringThrow();
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    try {
        System.out.print(*throw[throw]+=throw => ");
        doublesThrow()[indexThrow()] += doubleThrow();
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    try {
        System.out.print(*throw[throw]+=\"heh\" => ");
        stringsThrow()[indexThrow()] += "heh";
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    try {
        System.out.print(*throw[throw]+=12345 => ");
        doublesThrow()[indexThrow()] += 12345;
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    try {
        System.out.print(*throw[1]+=throw => ");
        stringsThrow()[1] += stringThrow();
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    try {
        System.out.print(*throw[1]+=throw => ");
        doublesThrow()[1] += doubleThrow();
        System.out.println("Okay!");
    }
}

```

```

    } catch (Throwable e) { System.out.println(name(e)); }
    try {
        System.out.print("throw[1]+=\"heh\" => ");
        stringsThrow()[1] += "heh";
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    try {
        System.out.print("throw[1]+=12345 => ");
        doublesThrow()[1] += 12345;
        System.out.println("Okay!");
    } catch (Throwable e) { System.out.println(name(e)); }
    testEight(null, null, 1);
    testEight(null, null, 9);
    testEight(strings, doubles, 1);
    testEight(strings, doubles, 9);
}
}

```

此程序输出:

```

throw[throw]+=throw => ArrayReferenceThrow
throw[throw]+=throw => ArrayReferenceThrow
throw[throw]+="heh" => ArrayReferenceThrow
throw[throw]+=12345 => ArrayReferenceThrow
throw[1]+=throw => ArrayReferenceThrow
throw[1]+=throw => ArrayReferenceThrow
throw[1]+="heh" => ArrayReferenceThrow
throw[1]+=12345 => ArrayReferenceThrow

null[throw]+=throw => IndexThrow
null[throw]+=throw => IndexThrow
null[throw]+="heh" => IndexThrow
null[throw]+=12345 => IndexThrow
null[1]+=throw => NullPointerException
null[1]+=throw => NullPointerException
null[1]+="heh" => NullPointerException
null[1]+=12345 => NullPointerException

null[throw]+=throw => IndexThrow
null[throw]+=throw => IndexThrow
null[throw]+="heh" => IndexThrow
null[throw]+=12345 => IndexThrow
null[9]+=throw => NullPointerException
null[9]+=throw => NullPointerException
null[9]+="heh" => NullPointerException
null[9]+=12345 => NullPointerException

```

```

Strings[throw] += throw => IndexThrow
doubles[throw] += throw => IndexThrow
Strings[throw] += "heh" => IndexThrow
doubles[throw] += 12345 => IndexThrow
Strings[1] += throw => RightHandSideThrow
doubles[1] += throw => RightHandSideThrow
Strings[1] += "heh" => Okay!
doubles[1] += 12345 => Okay!

Strings[throw] += throw => IndexThrow
doubles[throw] += throw => IndexThrow
Strings[throw] += "heh" => IndexThrow
doubles[throw] += 12345 => IndexThrow
Strings[9] += throw => ArrayIndexOutOfBoundsException
doubles[9] += throw => ArrayIndexOutOfBoundsException
Strings[9] += "heh" => ArrayIndexOutOfBoundsException
doubles[9] += 12345 => ArrayIndexOutOfBoundsException

```

程序中最有意思的情形是第倒数 10 和 11 种情形：

```

Strings[1] += throw => RightHandSideThrow
doubles[1] += throw => RightHandSideThrow

```

它们是这样的情形，即抛出异常的右边真正开始抛出异常；而且，它们是仅有的一些幸运的情形。这展示了右边操作数的求值确实在检查空数组引用值和超出边界索引值之后发生。

下面的程序展示了一个事实，即组合赋值的左边的值在对右边求值前进行了保存：

```

class Test {
    public static void main(String[] args) {
        int k = 1;
        int[] a = { 1 };
        k += (k = 4) * (k + 2);
        a[0] += (a[0] = 4) * (a[0] + 2);
        System.out.println("k==" + k + " and a[0]==" + a[0]);
    }
}

```

此程序输出：

k==25 和 a[0]==25

k 的值 1 在计算它的右边操作数  $(k = 4) * (k + 2)$  前被元素赋值运算符 += 保存。然后它的右边操作数的求值将 4 赋予 k，为  $k + 2$  计算值 6，然后 4 乘以 6 得 24。然后将它添加到保存值 1 得 25，之后该值通过 += 运算符存储到 k。一个相等的分析应用于使用 a[0] 的情形。简言之，语句：

```
k += (k = 4) * (k + 2);
```

```
a[0] += (a[0] = 4) * (a[0] + 2);
```

正好像下面的语句以相同的方式进行表现：

```
k = k + (k = 4) * (k + 2);  
a[0] = a[0] + (a[0] = 4) * (a[0] + 2);
```

## 15.27 表达式

表达式是任何一个赋值表达式：

*Expression:*

*AssignmentExpression*

与 C 和 C++ 不同，Java 编程语言没有逗号运算符。

## 15.28 常量表达式

……年老的和专注的表情是固定的，不是一件偶然的事……  
——查尔斯·狄更斯，《双城记》(1859)

*ConstantExpression:*

*Expression*

编译时常量表达式是一个表示基本类型或 String 的值的表达式，该表达式不会突然结束，并只使用下面的内容组成：

- 基本类型文本和类型为 String 的文本 (3.10.5 节)
  - 强制转换到基本类型和强制转换到类型 String
  - 一元运算符 +、-、~ 和 ! (但不是 ++ 或 --)
  - 乘运算符 \*、/ 和 %
  - 加运算符 + 和 -
  - 移位运算符 <<、>> 和 >>>
  - 关系运算符 <、<=、> 和 >= (但不是 instanceof)
  - 相等运算符 == 和 !=
  - 位和逻辑运算符 &、^ 和 |
  - 条件与运算符 && 和条件或运算符 ||
  - 三重条件运算符 ? :
  - 其包含的表达式是一个常量表达式的带括号表达式
  - 引用常量变量 (4.12.4 节) 的简单名称
  - 形式为 *TypeName* . *Identifier* 的引用常量变量 (4.12.4 节) 的限定名称
- 编译时常量表达式用在 switch 语句中的 case 标签中，并且对于赋值转换 (5.2 节)

有一个特殊的含义。类型为 `String` 的编译时常量始终是“interned”，以便使用方法 `String.intern` 共享惟一的一些实例。

编译时常量表达式始终被认为是精确浮点（15.4 节）的，即使它出现这样的情形中：非常量表达式不会被认为是精确浮点的。

下面是常量表达式的一些例子：

```
true
(short) (1*2*3*4*5*6)
Integer.MAX_VALUE / 2
2.0 * Math.PI
"The integer" + Long.MAX_VALUE + " is mighty big."
```

当群众的脸转向他并且含糊的眼睛注视他时，他装出最浪漫的表情……

——F.Scott Fitzgerald, 《This Side of Paradise》(1920)

# 第 16 章

## 明确赋值

我们已知的所有进化都是从模糊到明确。

——Charles Peirce

无论何时何地以何种方式访问局部变量(14.4节)和空白 `final`(4.12.4节)字段(8.3.1.2节)，它们都必须明确赋值。对该值的访问由变量的简单名组成，它能够出现在表达式中的任意位置，但简单的赋值运算符(=)的左操作数除外。因此，Java 编译器必须执行细致稳健的流分析，来确保 `f` (局部变量或空白 `final` 字段)在被访问前已明确赋值，否则必须告知发生编译时错误。

类似地，每个空白 `final` 变量至多赋值一次。在为其赋值时，它必须是明确未赋值。这种赋值被定义为：当且仅当变量的简单名或 `this` 限定的简单名出现在赋值运算符左侧时才发生赋值。因此，Java 编译器必须执行细致稳健的流分析，来确保每次为空白 `final` 变量赋值时，该变量在赋值前明确未赋值，否则必须告知发生编译时错误。

本章剩余部分将重点解释“前明确赋值”和“前明确未赋值”两个概念。

明确赋值背后的思想是：局部变量或空白 `final` 字段的赋值必须出现在访问它们的每条可能的执行路径上。同样，明确未赋值背后的思想是：只允许空白 `final` 字段有一个赋值操作出现在进行该赋值的任何可能的执行路径上。相应的分析会考虑语句和表达式的结构，并且同时还提供对表达式运算符 `!`、`&&`、`||`、`?`、`:` 和布尔值常量表达式的特殊处理。

例如，在以下代码中，Java 编译器会识别对 `k` 访问（作为方法调用的参数）之前已明确赋值：

```
{
    int k;
    if (v > 0 && (k = System.in.read()) >= 0)
        System.out.println(k);
}
```

因为仅当以下表达式值为真时，访问才会发生：

```
v > 0 && (k = System.in.read()) >= 0
```

而且仅当对 `k` 执行了赋值，该表达式值才可能为真（可以更恰当地估算）。



类似地，在以下代码中，Java 编译器将通过 while 语句识别变量 k 已明确赋值：

```
{
    int k;
    while (true) {
        k = n;
        if (k >= 5) break;
        n = 6;
    }
    System.out.println(k);
}
```

因为条件表达式 true 永远都不会是 false，所以只有 break 语句才能引发 while 语句正常完成，而 k 已在 break 语句前明确赋值。

从另一方面看，以下代码：

```
{
    int k;
    while (n < 4) {
        k = n;
        if (k >= 5) break;
        n = 6;
    }
    System.out.println(k); // k is not "definitely assigned" before this
}
```

肯定会被 Java 编译器拒绝，因为在这种情况下，按照明确赋值规则，while 语句无法保证会执行其程序体。

除了条件布尔运算符 &&、||、? : 和布尔值常量表达式的特定处理外，流分析不会考虑表达式值。

例如，Java 编译器必须为以下代码生成编译时错误：

```
{
    int k;
    int n = 5;
    if (n > 2)
        k = 3;
    System.out.println(k); // k is not "definitely assigned" before this
}
```

即使 n 的值在编译时已知，并且从原则上讲，在编译时可以知道对 k 进行赋值将始终执行（可以更恰当地估算）。Java 编译器必须根据本节规则运行。该规则仅识别常量表达式。在本例中，表达式  $n > 2$  不是第 15.28 节中所定义的常量表达式。

再如，按照 k 的明确赋值而言，Java 编译器将接受以下代码：

```
void flow(boolean flag) {
    int k;
    if (flag)
        k = 3;
    else
```

```
        k = 4;
        System.out.println(k);
    }
```

这是因为 `flag` 无论是 `true` 或 `false`，本节概述的规则都能够判明 `k` 已赋值。但是该规则不接受以下变种：

```
void flow(boolean flag) {
    int k;
    if (flag)
        k = 3;
    if (!flag)
        k = 4;
    System.out.println(k); //k is not "definitely assigned" before here
}
```

因此编译这个程序必会发生编译时错误。

下面一个相关示例说明了明确未赋值规则。就 `k` 的明确未赋值而言，Java 编译器将接受以下代码：

```
void unflow(boolean flag) {
    final int k;
    if (flag) {
        k = 3;
        System.out.println(k);
    }
    else {
        k = 4;
        System.out.println(k);
    }
}
```

这是因为无论 `flag` 是 `true` 或 `false`，本节概述的规则都能够判明 `k` 至多赋值一次（实际上只有一次）。但是该规则不接受以下变种：

```
void unflow(boolean flag) {
    final int k;
    if (flag) {
        k = 3;
        System.out.println(k);
    }
    if (!flag) {
        k = 4; //k is not "definitely unassigned" before here
        System.out.println(k);
    }
}
```

因此编译此程序必会引发编译时错误。

为了确切地指出“明确赋值”的所有情况，本节规则定义了以下几个技术术语：

- 变量是否在语句或表达式前明确赋值；

- 变量是否在语句或表达式前明确未赋值;
- 变量是否在语句或表达式后明确赋值;
- 变量是否在语句或表达式后明确未赋值。

对于布尔值表达式, 最后两条技术术语可以分成下列四种情况:

- 表达式为真时, 变量是否为后明确赋值;
- 表达式为真时, 变量是否为后明确未赋值;
- 表达式为假时, 变量是否为后明确赋值;
- 表达式为假时, 变量是否为后明确未赋值。

这里的为真和为假是指表达式的值。

例如, 当下列表达式为 true 而非 false 时[因为如果 a 为假, 那么对 k 的赋值就没必要执行(可以更恰当地估算)], 局部变量 k 在估算该表达式后被明确赋值:

```
a && ((k=m) > 5)
```

短语“v 在 x 后明确赋值”(这里的 v 是局部变量, x 是语句或表达式)的意思是“如果 x 正常完成, 则 v 在 x 后明确赋值”。而如果 x 突然结束, 那么赋值无需已发生, 这里声明的规则会考虑这个情况。该定义的特殊结果是: “v 在 break; 后明确赋值”永远为真! 这是因为 break 语句永远不会正常完成, 如果 break 语句正常完成, v 已赋值是空虚的真。

类似地, 语句“v 在 x 后明确未赋值”(这里的 v 是变量, x 是语句或表达式)的意思是“如果 x 正常完成, 那么 v 在 x 后明确未赋值”。该定义的结果更加特殊: “v 在 break; 后明确未赋值”始终为真! 这是因为 break 语句永远不会正常完成, 如果 break 语句正常完成, v 仍未赋值是空虚的真(对于这种情况, 如果 break 语句能够正常完成, 那么再不可能的事情也虚假的真了)。

变量 v 在执行语句或表达式后, 总共会出现以下 4 种可能性:

- v 明确赋值, 且没有明确未赋值。  
(流分析规则证明已为 v 赋值。)
- v 明确未赋值, 且没有明确赋值。  
(流分析规则证明还未为 v 赋值。)
- v 没有明确赋值, 且没有明确未赋值。  
(规则不能证明是否已为 v 赋值。)
- v 明确赋值, 且明确未赋值。  
(语句或表达式不可能正常完成。)

我们还使用缩写惯例: 如果规则出现一个或多个“[未]赋值”, 那么它代表两个规则, 一种是由“明确赋值”代替“[未]赋值”出现, 一种是由“明确未赋值”代替“[未]赋值”出现。

例如:

- v 在空语句之后“[未]赋值”, 当且仅当其在空语句之前“[未]赋值”。

上面这句话应该理解为代表下面两种规则:

- v 在空语句之后明确赋值, 当且仅当其在空语句之前明确赋值。

- $v$  在空语句之后明确未赋值，当且仅当其在空语句之前明确未赋值。

循环语句的明确未赋值分析引出了一个特殊问题。请考虑 `while (e) S` 语句。为了确定  $v$  是否在  $e$  的一些子表达式内明确未赋值，我们需要确定  $v$  是否在  $e$  之前明确未赋值。根据明确赋值的规则（16.2.10 节）类推，一种观点认为  $v$  在  $e$  之前明确未赋值，当且仅当其在 `while` 语句之前明确未赋值。然而，对于我们的目的来说，这样的规则是不充分的。如果估算  $e$  为真，那么将执行语句  $S$ 。然后，如果  $v$  由  $S$  赋值，则在估算  $e$  时后面的迭代(s)中  $v$  将被赋值。根据上面建议的规则，有可能对  $v$  进行多次赋值，而这正是我们通过引入这些规则，所要力求避免的。

修改后的规则是：“ $v$  在  $e$  之前明确未赋值，当且仅当在 `while` 语句之前明确未赋值，并且在  $S$  之后明确未赋值”。然而，当我们为  $S$  公式化表述时，我们发现：“ $v$  在  $S$  之前明确未赋值，当且仅当其在  $e$  为真后明确未赋值”。但这导致了一个循环。实际上， $v$  在循环条件  $e$  之前明确未赋值，当且仅当其作为整体在循环后未赋值！

我们通过对循环条件和循环体进行假设分析，来中断这种恶性循环。例如，如果我们假设  $v$  在  $e$  之前明确未赋值（无论  $v$  是否真正地在  $e$  之前明确未赋值），然后可以证明  $v$  在  $e$  之后明确未赋值，那么我们知道  $e$  不会赋值  $v$ 。下面是更为正式的叙述：

假设  $v$  在  $e$  之前明确未赋值，在  $e$  之后亦明确未赋值。

对于 Java 语言中的所有循环语句，上面所分析的变种常用来定义有充分依据的明确未赋值规则。

在本章的剩余部分，除非具有明确声明，否则我们将用  $v$  代表局部变量或空白 `final` 字段（对于明确赋值规则）或空白 `final` 变量（对于明确未赋值规则）。同样地，我们将使用  $a$ 、 $b$ 、 $c$  和  $e$  代表表达式，以及用  $S$  和  $T$  代表语句。用短语“ $a$  是  $v$ ”来代表  $a$  是变量  $v$  的简单名或者由 `this`（忽略圆括号）限定的  $v$  的简单名。用短语“ $a$  不是  $v$ ”来代表“ $a$  是  $v$ ”的相反情况。

## 16.1 明确赋值和表达式

*Driftwood*: 第一方当事人应该在合同中明确为甲方。

——Groucho Marx, 《歌声俪影》(1935)

### 16.1.1 布尔常量表达式

- 当值为 `true` 的常量表达式为假时， $v$  在该表达式之后[未]赋值。
- 当值为 `false` 的常量表达式为真时， $v$  在该表达式之后[未]赋值。

因为值为 `true` 的常量表达式永远都不会具有 `false` 值，值为 `false` 的常量表达式永远都不会具有 `true` 值，因此上面两种规则毫无意义地满足。但它们有助于分析涉及以下运算符的表达式——包括运算符 `&&`（16.1.2 节）、`||`（16.1.3 节）、`!`（16.1.4 节）、和 `?:`（16.1.5 节）。

- 当值为 `true` 的常量表达式为真时， $v$  在该表达式之后[未]赋值，当且仅当  $v$  在常量表达式之前[未]赋值。
- 当值为 `false` 的常量表达式为假时， $v$  在该表达式之后[未]赋值，当且仅当在该常

量表达式之前[未]赋值。

- $v$  在布尔值常量表达式  $e$  之后[未]赋值, 当且仅当  $v$  在  $e$  为真之后[未]赋值, 以及  $v$  在  $e$  为假之后[未]赋值(这等于是说  $v$  在  $e$  之后[未]赋值, 当且仅当  $v$  在  $e$  之前[未]赋值)。

### 16.1.2 布尔运算符 &&

- $v$  在  $a \&\& b$  为真之后[未]赋值, 当且仅当  $v$  在  $b$  为真之后[未]赋值。
- $v$  在  $a \&\& b$  为假之后[未]赋值, 当且仅当  $v$  在  $a$  为假之后[未]赋值, 并且  $v$  在  $b$  为假之后[未]赋值。
- $v$  在  $a$  之前[未]赋值, 当且仅当  $v$  在  $a \&\& b$  之前[未]赋值。
- $v$  在  $b$  之前[未]赋值, 当且仅当  $v$  在  $a$  为真之后[未]赋值。
- $v$  在  $a \&\& b$  之后[未]赋值, 当且仅当  $v$  在  $a \&\& b$  为真之后[未]赋值, 并且  $v$  在  $a \&\& b$  为假之后[未]赋值。

### 16.1.3 布尔运算符 ||

- $v$  在  $a || b$  为真之后[未]赋值, 当且仅当  $v$  在  $a$  为真之后[未]赋值, 并且  $v$  在  $b$  为真之后[未]赋值。
- $v$  在  $a || b$  为假之后[未]赋值, 当且仅当  $v$  在  $b$  为假之后[未]赋值。
- $v$  在  $a$  之前[未]赋值, 当且仅当  $v$  在  $a || b$  之前[未]赋值。
- $v$  在  $b$  之前[未]赋值, 当且仅当其在  $a$  为假之后[未]赋值。
- $v$  在  $a || b$  之后[未]赋值, 当且仅当  $v$  在  $a || b$  为真之后[未]赋值, 并且  $v$  在  $a || b$  为假之后[未]赋值。

### 16.1.4 布尔运算符 !

- $v$  在  $!a$  为真之后[未]赋值, 当且仅当  $v$  在  $a$  为假之后[未]赋值。
- $v$  在  $!a$  为假之后[未]赋值, 当且仅当  $v$  在  $a$  为真之后[未]赋值。
- $v$  在  $a$  之前[未]赋值, 当且仅当  $v$  在  $!a$  之前[未]赋值。
- $v$  在  $!a$  之后[未]赋值, 当且仅当  $v$  在  $!a$  为真之后[未]赋值, 并且  $v$  在  $!a$  为假之后[未]赋值(这就是说  $v$  在  $!a$  之后[未]赋值, 当且仅当  $v$  在  $a$  之后[未]赋值)。

### 16.1.5 布尔运算符 ? :

假设  $b$  和  $c$  是布尔值表达式。

- $v$  在  $a ? b : c$  为真之后[未]赋值, 当且仅当  $v$  在  $b$  为真之后[未]赋值, 并且  $v$  在  $c$  为真之后[未]赋值。
- $v$  在  $a ? b : c$  为假之后[未]赋值, 当且仅当  $v$  在  $b$  为假之后[未]赋值, 并且  $v$  在  $c$  为假之后[未]赋值。

- $v$  在  $a$  之前[未]赋值, 当且仅当  $v$  在  $a ? b : c$  之前[未]赋值。
- $v$  在  $b$  之前[未]赋值, 当且仅当  $v$  在  $a$  为真之后[未]赋值。
- $v$  在  $c$  之前[未]赋值, 当且仅当  $v$  在  $a$  为假之后[未]赋值。
- $v$  在  $a ? b : c$  之后[未]赋值, 当且仅当  $v$  在  $a ? b : c$  为真之后[未]赋值, 并且  $v$  在  $a ? b : c$  为假之后[未]赋值。

### 16.1.6 条件运算符 $?:$

假设  $b$  和  $c$  是非布尔值表达式。

- $v$  在  $a ? b : c$  之后[未]赋值, 当且仅当  $v$  在  $b$  之后[未]赋值, 并且  $v$  在  $c$  之后[未]赋值。
- $v$  在  $a$  之后[未]赋值, 当且仅当  $v$  在  $a ? b : c$  之前[未]赋值。
- $v$  在  $b$  之前[未]赋值, 当且仅当  $v$  在  $a$  为真之后[未]赋值。
- $v$  在  $c$  之前[未]赋值, 当且仅当  $v$  在  $a$  为假之后[未]赋值。

### 16.1.7 布尔类型的其他表达式

假设  $e$  是布尔类型的表达式且不是布尔常量表达式, 逻辑补表达式  $!a$ , 条件与表达式  $a \&\& b$ , 条件或表达式  $a || b$ , 以及条件表达式  $a ? b : c$ 。

- $v$  在  $e$  为真之后[未]赋值, 当且仅当  $v$  在  $e$  之后[未]赋值。
- $v$  在  $e$  为假之后[未]赋值, 当且仅当  $v$  在  $e$  之后[未]赋值。

### 16.1.8 赋值表达式

*Driftwood*: 您想要再听一次吗?

*Fiorello*: 希望解释一下第一方。

*Driftwood*: 您是什么意思? 甲方?

*Fiorello*: 不是, 甲方的第一方。

——Groucho Marx and Chico Marx, 《歌声倒影》(1935)

请考虑后面的赋值表达式  $a = b$ 、 $a += b$ 、 $a -= b$ 、 $a *= b$ 、 $a /= b$ 、 $a \% = b$ 、 $a << = b$ 、 $a >> = b$ 、 $a >>> = b$ 、 $a \& = b$ 、 $a |= b$  或  $a \wedge = b$ 。

- $v$  在赋值表达式之后明确赋值, 当且仅当  $a$  是  $v$  或  $v$  在  $b$  之后明确赋值。
- $v$  在赋值表达式之后明确未赋值, 当且仅当  $a$  不是  $v$  并且  $v$  在  $b$  之后明确未赋值。
- $v$  在  $a$  之前[未]赋值, 当且仅当  $v$  在赋值表达式之前[未]赋值。
- $v$  在  $b$  之前[未]赋值, 当且仅当  $v$  在  $a$  之后[未]赋值。

请注意, 如果  $a$  是  $v$ , 并且  $v$  在复合赋值 (例如,  $a \& = b$ ) 之前[未]赋值, 那么将有必要生成编译时错误。上面叙述的明确赋值的第一条规则甚至为复合赋值表达式包含离析项“ $a$  是  $v$ ”, 不仅是简单的赋值, 因此在代码后面  $v$  被认为已明确赋值。包括离析项“ $a$  是  $v$ ”不会影响到二次判定——关于程序是否可接受或将产生编译时错误, 但是它会影响到代码中可



能有多少处被认为是错误的，并且在实践中可以提高错误报告的质量。为了上面所叙述的明确未赋值，在第一条规则中将类似声明应用在结合项“ $a$  不是  $v$ ”的包含上。

### 16.1.9 运算符 $++$ 和 $--$

- $v$  在  $++a$ 、 $--a$ 、 $a++$  或  $a--$  之后明确赋值，当且仅当  $a$  是  $v$  或  $v$  在运算符表达式之后明确赋值。
- $v$  在  $++a$ 、 $--a$ 、 $a++$  和  $a--$  之后明确未赋值，当且仅当  $a$  不是  $v$  并且  $v$  在操作数表达式之后明确未赋值。
- $v$  在  $a$  之前[未]赋值，当且仅当  $v$  在  $++a$ 、 $--a$ 、 $a++$  或  $a--$  之前[未]赋值。

### 16.1.10 其他表达式

*Driftwood*: 好的。这就是说，第一方当事人  
应该在合同中明确为甲方。在合同中应该明确.....  
——Groucho Marx, 《歌声倒影》(1935)

如果表达式不是布尔常量表达式，也不是前递加表达式  $++a$ 、前递减表达式  $--a$ 、后递加表达式  $a++$ 、后递减表达式  $a--$ 、逻辑补表达式  $!a$ 、条件与表达式  $a \&\& b$ 、条件或表达式  $a \parallel b$ 、条件表示式  $a ? b : c$  或者赋值表达式，那么可以运用以下规则：

- 如果表达式没有子表达式， $v$  在表达式之后[未]赋值，当且仅当  $v$  在表达式之前[未]赋值。这种情况适用于文本常量、名称、`this`（包括限定和无限定两种）、没有参数的无限定类实例创建表达式、初始化方法不包含表达式的初始化数组创建表达式、无限定的超类字段访问表达式、没有参数的命名方法调用以及没有参数的无限定超类方法调用。
- 如果表达式有子表达式， $v$  在表达式之后[未]赋值，当且仅当  $v$  在最右边的子表达式之后[未]赋值。

在断定变量  $v$  可以在方法调用后明确未赋值的背后，有一些微妙的原因。自身获取表面值和自身无法限定，这种断言不会总是为真，因为调用的方法可能会执行赋值。但是必须记住 Java 程序语言的目的，明确未赋值的概念只应用于空白 `final` 变量。如果  $v$  是空白 `final` 局部变量，那么只有其声明所属的方法才可以为  $v$  赋值。如果  $v$  是空白 `final` 字段，那么只有包含  $v$  声明的类的构造函数或初始化方法才可以为  $v$  赋值，没有方法可以为  $v$  赋值。最后，显式的构造函数调用（8.8.7.1 节）是特殊（16.9 节）处理的，虽然它们在句法上类似于包含方法的表达式语句。由于它们不是表达式语句，因此本节规则是不适用于显式构造函数调用。

对于表达式  $x$  最邻近的子表达式  $y$ ，当且仅当下列情形之一为真时， $v$  在  $y$  之前[未]赋值：

- $y$  是  $x$  最左邻近的子表达式，并且  $v$  在  $x$  之前[未]赋值。
- $y$  是二元运算符的右操作数，并且  $v$  在左操作数之后[未]赋值。

- $x$  是数组访问,  $y$  是括号内的子表达式, 并且  $v$  在括号之前子表达式之后[未]赋值。
- $x$  是主方法调用表达式,  $y$  是方法调用表达式中的第一个参数表达式, 并且  $v$  在计算目标对象的主表达式之后[未]赋值。
- $x$  是方法调用表达式或类实例创建表达式。  $y$  是参数表达式, 但并非第一个, 并且  $v$  在该参数表达式之后[未]赋值给  $y$  的左部。
- $x$  是限定类实例创建表达式,  $y$  是在类实例创建表达式中的第一个参数表达式, 并且  $v$  在计算该限定对象的主表达式之后[未]赋值。
- $x$  是数组实例创建表达式,  $y$  是维表达式, 但不是第一个, 并且  $v$  在维表达式后[未]赋值给  $y$  的左部。
- $x$  是通过数组初始器初始化的数组实例创建表达式。  $y$  是  $x$  中的数组初始器, 并且  $v$  在维度表达式后[未]赋值给  $y$  的左部。

## 16.2 明确赋值和语句

*Driftwood*: 第二方当事人在合同中应该明确为乙方。  
——Groucho Marx, 《歌声倒影》(1935)

### 16.2.1 空语句

- $v$  在空语句之后[未]赋值, 当且仅当其在空语句之前[未]赋值。

### 16.2.2 块

- 空白 `final` 成员字段  $v$  在块 (该块是  $v$  作用域中的任意方法的程序体) 之前明确赋值 (并且不是明确未赋值)。
- 局部变量  $v$  在块 (该块是声明  $v$  的构造函数、方法、实例初始化方法或静态初始化方法的程序体) 之前明确未赋值 (并且不是明确赋值)。
- 假设  $c$  是  $v$  的作用域内声明的类, 那么:
  - ◆  $v$  在块 (该块是在  $c$  中声明的任何构造函数、方法、实例初始化方法或静态初始化方法的程序体) 之前明确赋值, 当且仅当  $v$  在  $c$  的声明之前明确赋值。

请注意, 这里没有规则允许我们推出  $v$  在块 (该块是在  $c$  中声明的任何构造函数、方法、实例初始化方法或静态初始化方法的程序体) 之前明确未赋值。但是我们可以非正式地推出  $v$  不是在块 (该块是在  $c$  中声明的任何构造函数、方法、实例初始化方法或静态初始化方法的程序体) 之前明确未赋值, 但是在这里无需为该规则显式声明。

- $c$  在空块之后[未]赋值, 当且仅当其在空块之前[未]赋值。
- $v$  在非空块之后[未]赋值, 当且仅当其在块中的最后一条语句之后[未]赋值
- $v$  在块中的第一条语句之前[未]赋值, 当且仅当其在块之前[未]赋值
- $v$  在块的任何语句  $s$  之前[未]赋值, 当且仅当其在块中先于  $s$  最近处理的语句之后[未]赋值。

我们说  $V$  在块  $B$  的每个位置明确未赋值，当且仅当：

- $V$  在  $B$  之前明确未赋值。
- 在出现在  $B$  中的每个  $V = e$ 、 $V += e$ 、 $V -= e$ 、 $V *= e$ 、 $V /= e$ 、 $V \% = e$ 、 $V < < = e$ 、 $V > > = e$ 、 $V > > > = e$ 、 $V \& = e$ 、 $V | = e$  或  $V \wedge = e$  中的  $e$  之后， $V$  明确赋值。
- $V$  在每个出现在  $B$  中的表达式  $++V$ 、 $--V$ 、 $V++$  或  $V--$  之前明确赋值。

这些条件是不直观并且需要一些解释的。请考虑一个简单的赋值  $V = e$ 。如果  $V$  在  $e$  之后明确赋值，则会出现以下情形之一：

(1) 赋值出现在死代码中，并且  $V$  虚假地明确赋值。在这种情况下，赋值将不会真正发生，我们可以假设赋值表达式并未为  $V$  赋值。

(2)  $V$  在  $e$  之前已通过早前的表达式赋值。在这种情况下，当前赋值将产生一个编译时错误。

因此，我们可以推出如果程序符合条件将不会产生编译时错误，那么在  $B$  中对  $V$  的任何赋值都会在运行时发生。

### 16.2.3 局部类声明语句

- $V$  在局部类声明语句之后[未]赋值，当且仅当其在局部类声明语句之前[未]赋值。

### 16.2.4 局部变量声明语句

- $V$  在局部变量声明语句（该语句不包含变量初始化方法）之后[未]赋值，当且仅当其在局部变量声明语句之前[未]赋值。
- $V$  在局部变量声明语句（该语句至少包含一个变量初始化方法）之后明确赋值，当且仅当其在局部变量声明语句中的最后变量初始化方法之后明确赋值，或者该声明中的最后的变量初始化方法位于声明  $V$  的声明符中。
- $V$  在局部变量声明语句（该语句至少包含一个变量初始化方法）之后明确未赋值，当且仅当其在局部变量声明语句中的最后变量初始化方法之后明确未赋值，或者该声明中的最后的变量初始化方法没有位于声明  $V$  的声明符中。
- $V$  在局部变量声明语句中第一个变量初始化方法之前[未]赋值，当且仅当其在局部变量声明语句之前[未]赋值。
- $V$  在局部变量声明语句中的变量初始化方法  $e$ （而不是第一个变量初始化方法）之前明确赋值，当且仅当  $V$  在变量初始化方法之后明确赋值给  $e$  的左部，或者  $e$  左部的初始化表达式位于声明  $V$  的声明符中。
- $V$  在局部变量声明语句中的变量初始化方法  $e$ （而不是第一个变量初始化方法）之前明确未赋值，当且仅当  $V$  在变量初始化方法之后明确未赋值给  $e$  的左部，或者  $e$  左部的初始化表达式没有位于声明  $V$  的声明符中。

### 16.2.5 标签语句

- $V$  在标签语句  $L:S$ （这里的  $L$  是一个标签）之后[未]赋值，当且仅当  $V$  在  $S$  之后[未]

赋值, 并且  $v$  在每个 `break` 语句 (该语句可以退出标签语句  $L:S$ ) 之前[未]赋值。

- $v$  在  $S$  之前[未]赋值, 当且仅当  $v$  在  $L:S$  之前[未]赋值。

### 16.2.6 表达式语句

- $v$  在表达式语句  $e$  之后[未]赋值, 当且仅当其在  $e$  之后[未]赋值。
- $v$  在  $e$  之前[未]赋值, 当且仅当其在  $e$  之前[未]赋值。

### 16.2.7 if 语句

下列规则适用于语句 `if ( $e$ )  $S$`  中:

- $v$  已在 `if ( $e$ )  $S$`  之后[未]赋值, 当且仅当  $v$  在  $S$  之后[未]赋值, 并且在  $e$  为假之后[未]赋值。
- $v$  已在  $e$  之前[未]赋值, 当且仅当  $v$  在 `if ( $e$ )  $S$`  之前[未]赋值。
- $v$  已在  $S$  之前[未]赋值, 当且仅当  $v$  在  $e$  为真之后[未]赋值。

下列规则适用于语句 `if ( $e$ )  $S$  else  $T$` :

- $v$  在 `if ( $e$ )  $S$  else  $T$`  之后[未]赋值, 当且仅当  $v$  在  $S$  之后[未]赋值, 并且  $v$  在  $T$  之后[未]赋值。
- $v$  在  $e$  之前[未]赋值, 当且仅当  $v$  在 `if ( $e$ )  $S$  else  $T$`  之前[未]赋值。
- $v$  在  $S$  之前[未]赋值, 当且仅当  $v$  在  $e$  为真之后[未]赋值。
- $v$  在  $T$  之前[未]赋值, 当且仅当  $v$  在  $e$  为假之后[未]赋值。

### 16.2.8 assert 语句

下列规则适用于 `assert  $e1$`  语句和 `assert  $e1:e2$`  语句:

- $v$  在  $e1$  之前明确[未]赋值, 当且仅当  $v$  在 `assert` 语句之前明确[未]赋值。
- $v$  在 `assert` 语句之前明确赋值, 当且仅当  $v$  在 `assert` 语句之前明确赋值。
- $v$  在 `assert` 语句之后明确未赋值, 当且仅当  $v$  在 `assert` 语句之前明确未赋值, 并且在  $e1$  为真之后明确未赋值。

下列规则适用于语句 `assert  $e1:e2$` :

- $v$  在  $e2$  之前明确[未]赋值, 当且仅当  $v$  在  $e1$  为假之后明确[未]赋值。

### 16.2.9 switch 语句

- $v$  在 `switch` 语句之后[未]赋值, 当且仅当下列所有情况都为真:
  - ◆ `switch` 块中有一个 `default` 标签, 或  $v$  在 `switch` 表达式之后[未]赋值。
  - ◆ 在不以块-语句-群开始的 `switch` 块中没有 `switch` 标签 (就是说, `switch` 块的末尾 “}” 之前近邻的不是 `switch` 标签), 或者  $v$  在 `switch` 表达式之后[未]赋值。
  - ◆ `switch` 块不包含块-语句-群, 或者  $v$  在最后的块-语句-群的最后块-语句之后[未]赋值。

- ◆  $v$  在每个 `break` 语句（该语句可以退出 `switch` 语句）之前[未]赋值。
  - $v$  在 `switch` 表达式之前[未]赋值，当且仅当  $v$  在 `switch` 语句之前[未]赋值。
- 如果 `switch` 块至少包含块-语句-群，那么下列规则也适用：
- $v$  在 `switch` 块中第一个块-语句群的第一个块-语句之前[未]赋值，当且仅当  $v$  在 `switch` 表达式之后[未]赋值。
  - $v$  在任何块-语句群的第一个块-语句（第一个块-语句群除外）之前[未]赋值，当且仅当  $v$  在 `switch` 表达式之后[未]赋值，并且在前面的块-语句之后[未]赋值。

### 16.2.10 while 语句

- $v$  在 `while (e) S` 之后[未]赋值，当且仅当  $e$  为假时，并且  $v$  是后[未]赋值， $v$  已在每个 `break` 语句（由于 `while` 语句是 `break` 对象）之前[未]赋值。
- $v$  在  $e$  之前明确赋值，当且仅当  $v$  在 `while` 语句之前明确赋值。
- $v$  在  $e$  之前明确未赋值，当且仅当下列所有的条件都支持：
  - ◆  $v$  在 `while` 语句之前明确未赋值。
  - ◆ 假设  $v$  在  $e$  之前明确未赋值，那么  $v$  在  $S$  之后明确未赋值。
  - ◆ 假设  $v$  在  $e$  之前明确未赋值，那么  $v$  在每个 `continue` 语句（由于 `while` 语句是 `continue` 对象）之前明确未赋值。
- $v$  在  $S$  之前[未]赋值，当且仅当  $v$  在  $e$  为真之后[未]赋值。

### 16.2.11 do 语句

- $v$  在 `do S while (e)` 之后[未]赋值，当且仅当  $v$  在  $e$  为假之后[未]赋值，并且  $v$  在每个 `break` 语句（由于 `do` 语句是 `break` 目标）之前[未]赋值。
- $v$  在  $S$  之前明确赋值，当且仅当  $v$  在 `do` 语句之前明确赋值。
- $v$  在  $S$  之前明确未赋值，当且仅当下列所有条件都成立：
  - ◆  $v$  在 `do` 语句之前明确未赋值。
  - ◆ 假设  $v$  在  $S$  之前明确未赋值，那么  $v$  在  $e$  为真之后明确未赋值。
- $v$  在  $e$  之前[未]赋值，当且仅当  $v$  在  $S$  之后[未]赋值，并且  $v$  在每个 `continue` 语句（由于 `do` 语句是 `continue` 目标）之前[未]赋值。

### 16.2.12 for 语句

在这里，规则包含了基本的 `for` 语句（14.14.1 节）。自从通过翻译成基本的 `for` 语句来定义增强的 `for` 语句（14.14.2 节），就不需要特定的规则了。

- $v$  在一个 `for` 语句之后[未]赋值，当且仅当下列两种情形为真：
  - ◆ 条件表达式不存在，或  $v$  在条件表达式为假之后[未]赋值。
  - ◆  $v$  在每个 `break` 语句（由于 `for` 语句是 `break` 目标）之前[未]赋值。
- $v$  在 `for` 语句的初始化部分之前[未]赋值，当且仅当  $v$  在 `for` 语句之前[未]赋值。



- $v$  在 `for` 语句的条件部分之前明确赋值, 当且仅当  $v$  在 `for` 语句的初始化部分之后明确赋值。
- $v$  在 `for` 语句的条件部分之前明确未赋值, 当且仅当下列所有的条件都成立:
  - ◆  $v$  在 `for` 语句的初始化部分之后明确未赋值。
  - ◆ 假设  $v$  在 `for` 语句的条件部分之前明确未赋值, 那么  $v$  在条件语句之后明确未赋值。
  - ◆ 假设  $v$  在条件语句之前明确未赋值, 那么  $v$  在每个 `continue` 语句 (由于 `for` 语句是 `continue` 目标) 之前明确未赋值。
- $v$  在包含它的语句之前[未]赋值, 当且仅当下列之一为真:
  - ◆ 条件表达式存在, 并且  $v$  在条件表达式为真之后[未]赋值。
  - ◆ 没有条件表达式存在, 并且  $v$  在 `for` 语句的初始化部分之后[未]赋值。
- $v$  在 `for` 语句的增量部分之前[未]赋值, 当且仅当  $v$  在包含语句之后[未]赋值, 并且  $v$  在每个 `continue` 语句 (由于 `for` 语句是 `continue` 目标) 之前[未]赋值。

#### 16.2.12.1 初始化部分

- 如果 `for` 语句的初始化部分是局部变量声明语句, 则应用第 16.2.4 节的规则。
- 另外, 如果初始化部分是空, 那么当且仅当  $v$  在初始化部分之前[未]赋值,  $v$  在初始化部分之后[未]赋值。
- 用以下三条规则:
  - ◆ 在初始化部分之后[未]赋值, 当且仅当  $v$  在初始化部分中的最后的表达式语句之后[未]赋值。
  - ◆ 在初始化部分中的第一个表达式语句之前[未]赋值, 当且仅当  $v$  在表达式部分之前[未]赋值。
  - ◆ 在初始化部分中的表达式语句  $E$  (第一个表达式语句除外) 之前[未]赋值, 当且仅当  $v$  已直接在前面  $E$  的表达式语句之后[未]赋值。

#### 16.2.12.2 增量部分

- 如果 `for` 语句的增量部分为空, 那么  $v$  在增量部分之后[未]赋值, 当且仅当  $v$  在增量部分之前[未]赋值。
- 否则, 应用下列三条规则:
  - ◆  $v$  在增量部分之后[未]赋值, 当且仅当  $v$  在增量部分中的最后表达式之后[未]赋值。
  - ◆  $v$  在增量部分中的第  $i$  个表达式语句之前[未]赋值, 当且仅当  $v$  在增量部分之前[未]赋值。
  - ◆  $v$  在增量部分中的表达式语句  $E$  (第  $i$  个表达式语句除外) 之前[未]赋值, 当且仅当  $v$  在  $E$  前紧贴的表达式语句之后[未]赋值。

#### 16.2.13 `break`、`continue`、`return` 和 `throw` 语句

*Fiorello*: 那么, 为什么乙方的第一方不能是甲方的第二方呢? 那样您会得到什么?

——Chico Marx, 《歌声倒影》(1935)



- 依照规定, 我们说  $v$  在任何的 `break`、`continue`、`return` 或 `throw` 语句之后[未]赋值。该概念是变量为“后[未]赋值”语句或表达式, 其真正的意思“是在语句或表达式正常完成之后[未]赋值”。因为 `break`、`continue`、`return` 或 `throw` 语句从未正常完成过, 那么这种概念是虚假的。
- 在带有表达式  $e$  的 `return` 语句, 或带有表达式  $e$  的 `throw` 语句中,  $v$  在  $e$  之前[未]赋值, 当且仅当  $v$  在 `return` 或 `throw` 语句之前[未]赋值。

#### 16.2.14 `synchronized` 语句

- $v$  在 `synchronized (e) S` 之后[未]赋值, 当且仅当  $v$  在  $S$  之后[未]赋值。
- $v$  在  $e$  之前[未]赋值, 当且仅当  $v$  在语句 `synchronized (e) S` 之前[未]赋值。
- $v$  在  $S$  之前[未]赋值, 当且仅当  $v$  在  $e$  之后[未]赋值。

#### 16.2.15 `try` 语句

下列规则适用于每个 `try` 语句, 无论其是否具有 `finally` 块:

- $v$  在 `try` 块之前[未]赋值, 当且仅当  $v$  在 `try` 语句之前[未]赋值。
- $v$  在 `catch` 块之前明确赋值, 当且仅当  $v$  在 `try` 块之前明确赋值。
- $v$  在 `catch` 块之前明确未赋值, 当且仅当下列所有条件都支持:
  - ◆  $v$  在 `try` 块之后明确未赋值。
  - ◆  $v$  在属于 `try` 块的每个 `return` 语句之前明确未赋值。
  - ◆  $v$  在  $e$  (在属于 `try` 块的格式 `throw e` 的每个语句中) 之后明确未赋值。
  - ◆  $v$  在  $e1$  (由于格式 `assert e1` 的每个语句出现在 `try` 块中) 之后明确未赋值。
  - ◆  $v$  在  $e2$  (是在格式 `assert e1: e2` 的每个语句中, 并出现在 `try` 块中) 之后明确未赋值。
  - ◆  $v$  在属于 `try` 块并且其 `break` 目标包含 (或者就是) `try` 语句的每个 `break` 语句之前明确未赋值的。
  - ◆  $v$  在属于 `try` 块并且其 `continue` 目标包含 `try` 语句的每个 `continue` 语句之前明确未赋值的。

如果 `try` 语句没有 `finally` 块, 那么此规则也适用:

- $v$  是 `try` 语句之后[未]赋值, 当且仅当  $v$  是 `try` 块之后[未]赋值, 并且  $v$  在 `try` 语句中每个 `catch` 块后是未被赋值的。

如果 `try` 语句有 `finally` 块, 那么此规则也同样适用:

- $v$  在 `try` 语句之后赋值, 当且仅当下列规则之一为真:
  - ◆  $v$  在 `try` 块之后明确赋值, 并且  $v$  在 `try` 语句中的每个 `catch` 块之后明确赋值。
  - ◆  $v$  在 `finally` 块之后明确赋值。
  - ◆  $v$  在 `try` 语句之后明确未赋值, 当且仅当  $v$  在 `finally` 块之后明确未赋值。
- $v$  在 `finally` 块之前明确赋值, 当且仅当  $v$  在 `try` 语句之前明确赋值。
- $v$  在 `finally` 块之前明确未赋值, 当且仅当下列所有的条件都成立:

- ◆  $v$  在 `try` 块之后明确未赋值。
- ◆  $v$  在属于 `try` 块的每个 `return` 语句之前明确未赋值。
- ◆  $v$  在  $e$  (在属于 `try` 块的格式 `throw e` 的每个语句中) 之后明确未赋值。
- ◆  $v$  在  $e1$  (由于格式 `assert e1` 的每个语句出现在 `try` 块中) 之后明确未赋值。
- ◆  $v$  在  $e2$  (是在格式 `assert e1: e2` 的每个语句中并出现在 `try` 块中) 之后明确未赋值。
- ◆  $v$  在属于 `try` 块的每个 `break` 语句之前明确未赋值, 并且 `break` 语句包含 `try` 语句。
- ◆  $v$  在属于 `try` 块的每个 `continue` 语句之前明确未赋值, 并且 `continue` 语句包含 `try` 语句。
- ◆  $v$  是每个 `try` 的语句之后的明确未赋值。

### 16.3 明确赋值和参数

- 方法或构造函数的形参  $v$  在方法或构造函数的程序体之前明确赋值 (而且不是明确未赋值)。
- `catch` 子句的异常参数  $v$  在 `catch` 子句语体之前明确赋值 (而且不是明确未赋值)。

### 16.4 明确赋值和数组初始化方法

- $v$  在空数组初始化方法之后[未]赋值, 当且仅当是在空数组初始化方法之前[未]赋值。
- $v$  在非空数组初始化方法之后[未]赋值, 当且仅当是在数组初始化方法中的最后的变量初始化之后赋[未]值。
- $v$  在数组初始化方法的第变量初始化之前[未]赋值, 当且仅当是在数组初始化方法之前[未]赋值。
- $v$  在数组初始化方法的任何变量初始化  $i$  之前[未]赋值, 当且仅当是在数组初始化方法中的  $i$  左边的变量初始化之后[未]赋值。

### 16.5 明确赋值和枚举常量

我们将在第 16.8 节中学习规则决定变量在枚举常量之前什么时候是明确赋值, 什么时候是明确未赋值。



这是因为枚举常量基本上是静态 `final` 字段 (8.3.1.1 节, 8.3.1.2 节), 而且该字段初始化后具有类实例创建表达式 (15.9 节)。

- $v$  在  $v$  的范围内而不带参数的枚举常量的类语体声明之前明确赋值，当且仅当  $v$  在枚举常量之前明确赋值。
- $v$  在  $v$  的范围内声明带参数的枚举常量的类语体声明之前明确赋值，当且仅当  $v$  在枚举常量的最后参数表达式之后明确赋值。

在枚举常量的类程序体内的任何构造函数的明确赋值/未赋值状况是取决于类的常规规则。

假设  $y$  是枚举常量的参数，但不是第一个参数。因此：

- $v$  在  $y$  之前[未]赋值，当且仅当是在参数到  $y$  左边之后[未]赋值。

其他：

- $v$  在第一个参数到枚举常量之前[未]赋值，当且仅当是在枚举常量之前[未]赋值。

## 16.6 明确赋值和匿名类

- $v$  在匿名类声明（15.9.5 节）（该声明在  $v$  的范围内声明）之前明确赋值，当且仅当  $v$  在类实例创建表达式（该表达式声明匿名类）之后明确赋值。

**附注**

枚举常量暗含匿名类定义，在第 16.5 节部分的规则中声明。

## 16.7 明确赋值和成员类型

假设  $c$  为类，并且假设  $v$  为  $c$  的空白 `final` 成员字段。那么：

- $v$  在  $c$  的任何成员类型声明之前明确赋值（而且是不明确未赋值）。

假设  $c$  在  $v$  范围内声明为类。那么：

- $v$  在  $c$  的成员类型（8.5 节，9.5 节）声明之前明确赋值，当且仅当  $v$  在  $c$  的声明之前明确赋值。

## 16.8 明确赋值和静态初始化方法

假设  $c$  在  $v$  范围内声明为类。那么：

- $v$  在枚举常量或  $c$  的静态变量初始化之前明确赋值，当且仅当  $v$  在  $c$  的声明之前明确赋值。

注意，这里没有规则让我们推出  $v$  在静态变量初始化或枚举常量之前明确未赋值。但是我们可以非正式地推出  $v$  不是在  $c$  的任何静态变量初始化之前明确未赋值，因此不需要为该种规则详细声明。

假设  $c$  为类，并假设  $v$  为  $c$  的空白 `final` 成员字段并在  $c$  中声明，那么：

- $v$  在  $c$  的最左边的枚举常量、`static` 初始化或 `static` 变量初始化之前明确未赋

值（而且不是明确赋值）。

- $v$  在  $C$  的枚举常量、static 初始化或 static 变量初始化（最左边除外）之前[未]赋值，当且仅当  $v$  在  $C$  的初期的枚举常量、static 初始化或 static 变量初始化之后[未]赋值。

假设  $C$  为类，并假设  $v$  为  $C$  的空白 final 成员字段，并在  $C$  的超类中声明，那么：

- $v$  是  $C$  的每个枚举常量之前的明确赋值（而且也是明确未赋值）。
- $v$  是  $C$  的静态初始化方法的体的块之前的明确赋值（而且不是明确未赋值）。
- $v$  是  $C$  的每个静态可变初始化方法之前的明确赋值（而且不是明确未赋值）。

## 16.9 明确赋值、构造函数和实例初始化方法

假设  $C$  是  $v$  范围内声明的类，因此：

- $v$  在  $C$  的实例变量初始化方法之前明确赋值，当且仅当  $v$  在  $C$  的声明之前明确赋值。

注意，这里没有规则让我们推出  $v$  在实例变量初始化方法之前明确未赋值。但是我们可以非正式推出  $v$  不是在  $C$  的任何实例变量初始化方法之前明确未赋值，因此，不需要为该种规则详细声明。

假设  $C$  为类，并假设  $v$  为  $C$  的空白 final 非 static 成员字段，并在  $C$  中声明。那么：

- $v$  在  $C$  的最左边的实例初始化或  $C$  的实例变量初始化方法之前明确未赋值（而且不是明确赋值）。
- $v$  在  $C$  的实例初始化或  $C$  的实例变量初始化方法（最左边的除外）之前[未]赋值，当且仅当  $v$  在  $C$  的初期实例初始化或  $C$  的实例变量初始化方法之后[未]赋值。

下列规则在类  $C$  的构造函数内成立：

- $v$  在备用构造函数调用（8.8.7.1 节）之后明确赋值（而且不是明确未赋值）。
- $v$  在显式或隐式超类构造函数调用（8.8.7.1 节）之前明确未赋值。
- 如果  $C$  没有实例初始化或实例变量初始化方法，那么  $v$  已不会在显式或隐式超类构造函数调用之后明确赋值（而且是明确未赋值）。
- 如果  $C$  至少有实例初始化或实例变量初始化方法，那么  $v$  在显式的或隐式的超类构造函数调用之后[未]赋值，当且仅当  $v$  在  $C$  最右边的实例初始化或  $C$  的实例变量初始化方法之后[未]赋值。

假设  $C$  为类，并假设  $v$  为  $C$  的空白 final 成员字段，并在  $C$  的超类中声明。那么：

- $v$  在块（是构造函数的程序体或  $C$  的实例初始化）之前明确赋值（而且不是明确未赋值）。
- $v$  在  $C$  的每个实例变量初始化方法之前明确赋值（而且不是明确未赋值）。

## 线程和锁

他们的身影总出现在至僻陋居孤单寂寞，  
惟有求知之心炽烈，  
如此坚毅地承受思想的奴役；  
如此敏锐地抽丝拔茧！

——William Wordsworth, 《Monks and Schoolmen》(1822)

尽管前面章节的大多数讨论主要涉及每次只执行单个语句或表达式时代码的行为，也就是说，通过单个线程，每个 Java 虚拟机可以立即支持执行的许多线程。这些线程独立执行代码，这些代码操作在驻留在共享主内存中的值和对象上。线程可以通过下述 3 种方式得到支持：具备许多硬件处理器，将单个硬件处理器分成一些时间片，或将许多硬件处理器分成一些时间片。

线程由 Thread 类表示。用户创建线程的惟一方式是创建此类的一个对象；每个线程和这样的对象相关。当在相应的 Thread 对象上调用 start() 方法时，一个线程将启动。

线程的行为，特别是当不正确同步时，可能是混淆和违反直觉的。本章描述了多线程程序的语义；它包括了这样的一些规则：通过读取多个线程更新的共享内存可以看到哪些值。由于规范类似于不同硬件体系结构的内存模型，所以这些语义称为 Java 编程语言内存模型。当不产生混淆时，我们将这些规则简称为“内存模型”。

这些语义没有指示应该如何执行多线程程序。相反，它们描述了多线程程序被允许展示的行为。只生成允许的行为的任何执行策略是一个可接受的执行策略。

## 17.1 锁

Java 编程语言提供了线程间通信的多种机制。这些方法中最基本的是同步化，此方法是使用监视器实现的。Java 中每个对象与一个监视器相关联，一个线程可以加锁和解锁此监视器。一次仅有一个线程可能在监视上持有锁。尝试锁住该监视器的任何其他线程被阻塞，直到它们可以在该监视器上获得一个锁。线程 *t* 可以多次锁住特别的监视器；每个解锁将一个加锁操作的作用反转来了。



synchronized 语句 (14.19 节) 计算了到一个对象的引用; 然后它尝试在该对象的监视器上执行加锁操作, 并不进一步继续, 直到锁操作已经成功完成。在加锁操作被执行完后, 会执行 synchronized 语句体。如果语句体的执行没有完成 (正常或突然), 那么将在相同的监视器上自动执行相同的解锁操作。

当调用 synchronized 方法 (8.4.3.6 节) 时, 该方法自动执行加锁操作: 在加锁操作成功完成前, 方法体没有被执行。如果该方法是一个实例方法, 它就锁住与调用它的实例 (也是在方法体执行期间将被称为 this 的对象) 有关的监视器。如果该方法是静态的, 它锁住与 Class 对象有关的监视器, Class 对象表示了定义方法的类。如果方法的体执行没有完成 (正常或突然), 将在相同的监视器上自动执行解锁操作。

Java 编程语言没有防止也没有要求检查死锁条件。线程在多个对象上 (直接或间接) 持有锁的程序应该使用传统技术来避免死锁, 创建不会死锁的高级加锁原语 (如果有必要的话)。

其他机制 (比如读写 java.util.concurrent 包中的 volatile 变量和类) 提供了一些同步的其他方法。

17.2 示例中的符号

这里指出的内存模型根本上不是基于 Java 编程语言的面向对象特征的。在我们的例子中, 出于简洁性和简单性考虑, 我们通常展示没有类或方法定义, 或显式废弃的代码段。大多数示例由两个或多个线程组成, 线程中包含一些语句用于访问本地变量、共享全局变量或对象的实例字段。我们通常使用变量名 r1 或 r2 来指出对于方法或线程是本地的变量。这样的一些变量不是可由其他线程方法访问的。

偏序和函数的限制。我们使用  $f|_d$  表示通过将  $f$  的域限制到  $d$  给出的函数: 对于  $d$  中的所有  $x$ ,  $f|_d(x)=f(x)$ , 对于不在  $d$  中的所有  $x$ ,  $f|_d(x)$  是未定义的。同样, 我们使用  $p|_d$  表示从偏序  $p$  到  $d$  中元素的限制: 对于  $d$  中所有  $x, y$ , 当且仅当  $p|_d(x, y)=p(x, y)$  时有  $p(x, y)$ 。如果任何  $x$  或  $y$  不在  $d$  中, 那就不是  $p|_d(x, y)$  的情形。

17.3 不正确同步的程序出现意外行为

Java 编程语言的语义允许编译器和微处理器执行优化, 这些优化能够以可以产生看似自相矛盾的过程行为的方式与不正确的同步代码进行交互。

跟踪 17.1: 重排语句生成的惊奇的结果——原始代码

线程 1	线程 2
1: r2=A;	3: r1=B;
2: B=1;	4: A=2;

跟踪 17.2: 重排语句产生的惊奇的结果——有效的编译器转换



线程 1	线程 2
B=1; r2=A;	r1=B; A=2;

例如，考虑跟踪 17.1 中的示例。此程序使用本地 `r1` 和 `r2` 和共享变量 `A` 和 `B`。最初，`A==B==0`。

出现结果 `r2==2`，`r1==1` 是可能的。直观而言，在执行中，指令 1 或指令 3 应该首先到来。如果指令 1 首先到来，它应该不能在指令 4 看到写。如果指令 3 首先到来，它应该不能在指令 2 看到写。

如果某个执行表现这种行为，那么我们将知道指令 4 在指令 1 之前到来，指令 1 在指令 2 之前到来，指令 2 在指令 3 之前到来，指令 3 在指令 4 之间到来。从表面判断，这是荒谬的。

但编译器被允许重排任一线程中的指令，同时这不会影响隔离中该线程的执行。如果指令 1 用指令 2 进行重排序，如跟踪 17.2 所示，那么就on易看到结果 `r2==2` 和 `r1==1` 如何发生的。

对于一些程序员，这种行为可以看成“中断”。但应该注意的是，此代码没有正确进行同步：

- 在一个线程中没有一个写
- 通过另一个线程读取相同的变量
- 及写和读没有通过同步进行排序

此情况是数据争用（17.4.5 节）的一个例子。当代码包含一个数据争用时，经常会产生违反直觉的结果。

几种机制可以产生跟踪 17.2 中的重新排序。实时编译器和处理器可以重新排列代码。此外，运行虚拟机所在体系结构的内存层次可能使它的出现像正被重排序的代码一样。在本章，我们将把可以重排序代码的任何内容称为编译器。

跟踪 17.3：向前替换产生的惊人结果

线程 1	线程 2
r1=p; r2=r1.x; r3=q; r4=r3.x; r5=r1.x;	r6=p; r6.x=3;

跟踪 17.4：向前替换产生的惊人结果

线程 1	线程 2
r1=p; r2=r1.x; r3=q; r4=r3.x; r5=r-2;	r6=p; r6.x=3;

惊人结果的另一个例子可以在跟踪 17.3 中看到。最初：`p==q`，`p.x==0`。此程序也没有正确地进行同步；它在没有在这些写之间强制任何排序的情况下写到了共享缓存。

一个常见的编译器优化包括将为 `r2` 读取的值重用于 `r5`：它们都是在没有干预写的情况下 `r1.x` 的读。跟踪 17.4 展示了这种情形。

现在考虑这样的情形：在线程 2 中赋值给 `r6.x` 碰巧发生在线程 1 中的 `r1.x` 的第一个读和 `r3.x` 的读之间。如果编译器决定将 `r2` 的值重用于 `r5`，那么 `r2` 和 `r5` 将有值 0，`r4` 将有值 3。从程序员的角度看，存储在 `p.x` 的值已经从 0 更改到 3，然后更改回来。

## 17.4 内存模型

给定一个程序和该程序的执行跟踪，内存模型描述了执行跟踪是否是该程序的一个合法执行。Java 程序语言内存模型通过下述方式工作：检查执行跟踪中的每个读，并根据某些规则检查该读观察的写是否是有效的。

内存模型描述了程序的可能行为。实现可以自由地产生任何它喜欢的代码，只要程序的所有结果执行产生可以由内存模型预测的结果。

### 讨论

这为实现者提供了大量的自由，以便执行无数的转换，包括操作的重新排序及不必要同步的删除。

内存模型决定了什么值可以在程序中的每个点读取。隔离中的每个线程的操作必须表现成由那个线程的语义控制，例外是每个读看到的值是由内存模型确定的。当我们提到这点时，我们说程序遵守线程内语义。线程内语义是单线程程序的语义，并允许基于线程中读操作看到的值完全预测线程的行为。要确定执行中线程 *t* 的操作是否是合法的，我们只要在线程 *t* 在单线程上下文中执行时评估它的实现，如该规范的其余部分所定义。

每次线程 *t* 的评估生成一个线程间操作，它必须匹配以程序顺序下一个到来的 *t* 的线程内操作 *a*。如果 *a* 是一个读，那么 *t* 的进一步评估使用内存模型定义的 *a* 看到的值。

本节提供了 Java 编程语言内存模型的规范，除了与 `final` 字段有关的问题，这些问题在第 17.5 节中描述。

### 17.4.1 共享变量

可以在线程之间共享的内存称为共享内存或堆内存。

所有的实例字段、静态字段和数组元素者存储在堆内存中。在本章，我们将使用术语变量来引用字段及数组元素。本地变量（14.4 节）、形式方法参数（8.4.1 节）或异常处理程序参数从未在线程之间共享，并不会受到内存模型的影响。

对同一个变量的两个方法（读取或写入）会产生冲突，前提是至少有一个访问是写。

## 17.4.2 操作

线程间操作是一个线程执行的操作，该线程可以由另一个线程检测或直接影响。有几种程序可以执行的进程间操作：

- 读（正常、或稳定）。读取一个变量。
- 写（正常、或稳定）。写一个变量。
- 同步操作，这些操作有：
  - ◆ 不稳定读。稳定读取一个变量。
  - ◆ 不稳定写。稳定写一个变量。
  - ◆ 加锁。锁住一个监视器。
  - ◆ 解锁。解锁一个监视器。
  - ◆ （综合）线程的第一个和最后一个操作。
  - ◆ 启动一个线程或检测线程是否已经终止的操作，如第 17.4.4 节所描述。
- 外部操作。外部操作中可以由执行的外部进行观察的动作，并且基于执行的外部环境有一个结果。
- 线程分歧操作（17.4.9 节）线程分歧操作只由无限循环中的线程执行，在此循环的过程中，没有内存、或外部操作被执行。如果线程执行了一个线程分歧操作，后面将有无数的线程分歧动作。

### 讨论

线程分歧操作被引入来模仿一个线程如何可以导致所有其他线程停止或进展失败。

此规范只与线程内操作有关。我们不需要关心线程内操作（比如添加两个局部变量和在第 3 个局部变量中存储结果）。如前所述，所有的线程需要遵守 Java 程序的正确线程内语义。通常，我们将更加简洁地将内线间操作称为简单操作。

动作  $a$  是由元组  $\langle t, k, v, u \rangle$  描述的，包括：

- $t$ ——执行操作的线程。
- $k$ ——操作的种类。
- $v$ ——操作中涉及的变量或监视器。对于加锁操作， $v$  是被加锁的监视器；对于解锁操作，它是正被解锁的监视器。如果动作是（不稳定或稳定）读，那么  $v$  就是正被读取的变量。如果动作是一个（不稳定或稳定）写，那么  $v$  就是一个正被写入的变量。
- 操作的任意惟一标识符。

一个外部操作元组包含有另一个元素，该元素包含可由执行操作的线程感知的外部操作。这可能是关于操作成功或失败的信息，或者动作读取的任何值。

外部操作的参数（比如哪些字节被写到哪个套接字）不是外部操作元组的一部分。这些参数是通过线程中的其他操作进行设置的，并可以通过检查线程内语义进行确定。

在非终结执行中，不是所有的外部操作都是可观察的。第 17.4.9 节中讨论了非终结执行和可观察操作。

### 17.4.3 程序和程序顺序

在每个线程  $t$  执行的所有线程内操作中,  $t$  的程序顺序是反映根据  $t$  的线程内语义会执行这些操作的顺序的总顺序。

如果所有的动作以与程序顺序一致的统一顺序(执行顺序)发生,那么一组动作就是连续一致的,并且,变量  $v$  的每个读  $r$  会看到写  $w$  写下的值,使得:

- 在执行顺序中,  $w$  在  $r$  之前到来,并且
- 在执行顺序中,没有其他写  $w'$  使得  $w$  在  $w'$  之前到来,  $w'$  在  $r$  之前到来。

连续一致性是一个非常强大的保证,该保证是关于程序执行中的可见性和顺序做出的。在连续一致性执行中,在所有单独动作(比如读和写)上有一个统一顺序,该顺序与程序的顺序一致,每个独立的操作是原子的,并且对于每个线程是立即可见的。

如果一个程序没有数据争用,那么程序的所有执行将表现为连续一致。

来自数据争用的连续一致性和/或自由仍然允许错误从操作集合产生,这些操作集合需要被自动感知,而不是不用被感知。

#### 讨论

如果要将连续一致性用作我们的内存模型,那么我们讨论的大多数编译器和处理器优化将是非法的。例如,在跟踪 17.3 中,一旦出现将 3 写到 `p.x`,后续对该位置的读会需要查看该值。

### 17.4.4 同步顺序

每个执行有一个同步顺序。同步顺序是执行的所有同步操作的统一顺序。对于每个线程  $t$ ,  $t$  中的同步操作(17.4.2 节)的同步顺序与  $t$  的程序顺序(17.4.3 节)是一致的。

同步操作在操作上归纳与...同步,如下定义:

- 监视器  $m$  上的解锁操作与  $m$  上的所有后续加锁操作同步(其中后续是根据同步顺序进行定义的)。
- 写到稳定变量(8.3.1.4 节)  $v$  与所有后续通过任何线程读取的  $v$  同步(其中后续是根据同步顺序定义的)。
- 启动线程的操作与它启动的线程中的第一个操作同步。
- 将默认值(0, false 或 null)写到每个变量与每个线程中的每个操作同步。尽管在分配包含变量的对象前,将默认值写到变量看起来有点奇怪,但从概念上讲,每个对象是在程序启动时用默认初始化值创建的。
- 线程  $T1$  中的最后操作与另一个线程  $T2$  中的任何操作同步,  $T2$  侦测  $T1$  是否终止。 $T2$  可以通过调用 `T1.isAlive()` 或 `T1.join()` 来完成这项工作。
- 如果  $T1$  中断了线程  $T2$ ,那么通过  $T1$  的中断就与任何其他线程(包括  $T2$ )确定  $T2$  是否已经被中断(通过抛出 `InterruptedException` 或通过调用 `Thread.interrupted` 或 `Thread.isInterrupted`)的任何点进行同步。

“与...同步”(synchronizes-with)边(edge)的来源称为释放,目的地称为获取。

### 17.4.5 之前发生顺序

两个操作可以通过之前发生关系进行排序。如果一个操作在另一个之前发生,然后第一个对于第二个是可见的,并且在第二个之前排序。

如果我们有二个操作  $x$  和  $y$ , 我们将写  $hb(x, y)$  来指出  $x$  在  $y$  之前发生。

- 如果  $x$  和  $y$  是同一个线程的操作,并且在程序顺序中  $x$  在  $y$  之前到来,那么  $hb(x, y)$ 。
- 有一条从对象的构造函数的结束到此对象的终结函数(12.6节)的之前发生边。
- 如果操作  $x$  与下面的操作  $y$  同步,那么我们也有  $hb(x, y)$ 。
- 如果有  $hb(x, y)$  和  $hb(y, z)$ , 那么有  $hb(x, z)$ 。

应该注意的是,两个操作之间存在的之前发生关系不一定表示它们必须按实现中的那个顺序发生。如果重新排序产生与合法执行一致的结果,那么它就不是非法的。

#### 讨论

例如,将默认值写到线程构造的对象的每个字段不需要在那个线程开始之前发生,只要没有读曾经观察到该事实。

更加特别地,如果两个操作共享一个之前发生关系,它们就不一定需要任何代码按那个顺序发生,借助这些代码,它们不需要共享一个之前发生关系。例如,与一个线程中的读进行数据争用的另一个线程中的写会对这些读表现为次序颠倒。

类 `Object` 的 `wait` 方法有与其相关的加锁和解锁机制;它们的之前发生关系是通过相关的操作定义的。这些方法在第 17.8 节中进一步描述。

之前发生关系定义了何时发生数据争用。

一组同步边  $S$  是足够的,前提是它是最小的集合,使得具有程序顺序的  $S$  的传递闭包确定了执行中所有的之前发生边。此集合是惟一的。

#### 讨论

从下面的定义推出:

- 监视器上的解锁在该监视器上的每个后续加锁之前发生。
- 写到稳定字段(8.3.1.4节)在该字段的每个后续读之前发生。
- 线程上对 `start()` 的调用在启用的线程中的任何操作之前发生。
- 线程中的所有操作在任何其他线程成功从该线程的 `join()` 中返回之前发生。
- 所有对象的默认初始化都在程序的所有其他操作(默认写除外)之前发生。

当程序包含通过之前发生关系排序的两个冲突访问(17.4.1节)时,就说它包含数据争用。

除了进程间操作之外的操作的语义,比如数组长度的读(10.7节),检查转换的执行



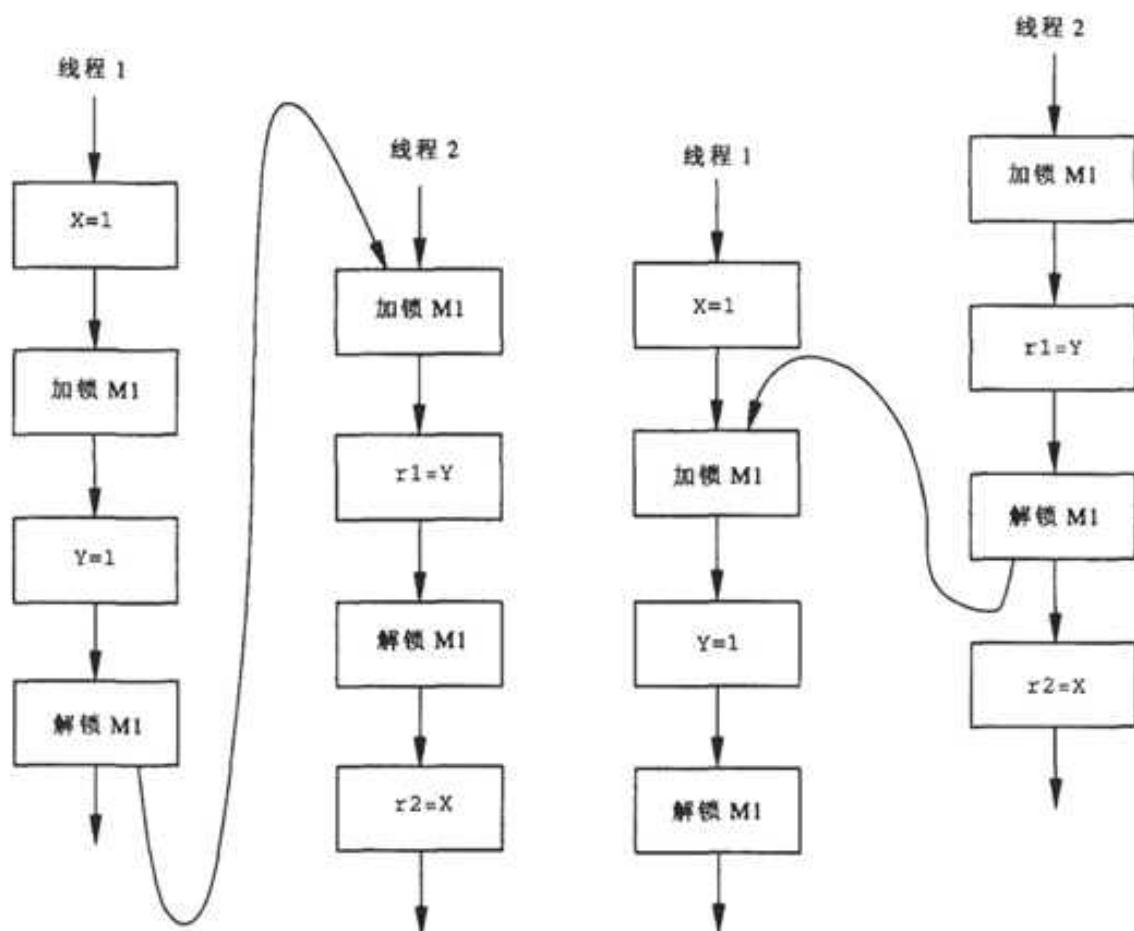
(5.5 节, 15.16 节) 及虚拟方法的调用 (15.12 节) 都没有直接受到数据争用的影响。

### 讨论

因此, 数据争用不会导致不正确的行为, 比如返回数据的错误长度。

当且仅当所有连续一致执行都没有数据争用时, 程序就是正确同步的。

不正确同步代码的一个微妙问题可以在下面看到。下面这些图展示了相同程序的两个不同执行, 两个执行都包含对共享变量  $x$  和  $y$  的冲突访问。程序中的两个线程加锁和解锁监视器  $M1$ 。在执行 (a) 中, 在所有冲突访问对之间有一个之前发生关系。但在执行 (b) 中, 在对  $x$  的冲突访问之间没有之前发生排序。因此, 程序不是正确同步的。



(a) 线程 1 首先获取锁: 对  $x$  的访问由之前发生进行排序

(b) 线程 2 首先获得锁: 对  $x$  的访问未通过之前发生进行排序

如果程序是正确同步的, 那么所有程序的执行将表现为连续一致的 (17.4.3 节)。



## 讨论

对于程序员，这是一个强有力的保证。程序员不需要知道重排序的原因，以确定他们的代码包含数据争用。因此，在确定他们的代码是否是正确同步的时候，他们不需要关于重排序的原因。一旦确定了代码是正确同步的，程序员就不需要担心重排序将影响他或她的代码。

程序必须是正确同步的，以避免各种反直觉行为，当代码被重新排序时，这些行为可以被观察到。正确同步的使用不能确保程序的全部行为是正确的。但它的使用不允许程序员以简单的方式推论有关程序的可能行为：正确同步程序的行为更不会依赖于可能的重排序。没有正确的同步，可能出现非常奇怪、混乱和违反直觉的行为。

如果在执行跟踪的之前发生偏序中，我们就说变量  $v$  的读  $r$  被允许观察到  $v$  的写。

- 在  $w$  之前  $r$  是没有排序的（也就是不是  $hb(r, w)$  的情形），并且
- 没有干预将  $w'$  写到  $v$  [也就是说，没有到  $v$  的写  $w'$  使得  $hb(w, w')$  和  $hb(w', r)$ ]。

非正式地，如果没有之前发生排序防止读  $r$ ，那么该读被允许看到写  $w$  的结果。

如果对于  $A$  中的所有读  $r$ ，它不是之前发生一致的，那么这操作集合  $A$  就是之前发生一致。

跟踪 17.5 之前发生一致性，但不是连接一致性允许的行为。可以观察  $r2==0$ ,  $r1==0$

线程 1	线程 2
$B=1;$	$A=2;$
$r2=A;$	$r1=B;$

情形是：要么  $hb(r, W(r))$ ，其中  $W(r)$  是由  $r$  看到的写操作，要么在  $A$  中存在着一个写，使得  $w.v=r.v$ 、 $hb(W(r), w)$  和  $hb(w, r)$ 。

## 讨论

在一组之前发生一致的操作中，每个读看到一个写，该写被允许通过之前发生排序进行查看。

例如，跟踪 17.5 中展示的行为是之前发生一致，因为有一些执行顺序允许每个读查看适当的写。

最初， $A==B==0$ 。在本例中，由于没有同步，所以每个读可以看到初始值的写或通过其他线程的写。一个这样的执行顺序是：

```
1: B = 1;
3: A = 2;
2: r2 = A; // sees initial write of 0
4: r1 = B; // sees initial write of 0
```

同样，跟踪 17.5 展示的行为是之前发生一致的，因为有一个执行顺序，该顺序允许每

个读看到适当的写。显示该行为的执行顺序是：

```
1: r2 = A; // sees write of A = 2
3: r1 = B; // sees write of B = 1
2: B = 1;
4: A = 2;
```

在本执行中，读看到了在执行顺序中后面发生的写。这可以视为违反直觉，但可通过之前发生一致性进行允许。允许读以后查看有时可能产生不可接受的行为的写。

### 17.4.6 执行

执行  $E$  是通过元组  $\langle P, A, po, so, W, V, sw, hb \rangle$  进行描述的，包括：

- $P$ ——一个程序
- $A$ ——一个操作集合
- $po$ ——程序顺序，对于每个线程  $t$ ，这是由  $A$  中的  $t$  执行的所有操作的统一顺序。
- $so$ ——同步顺序，它是  $A$  中所有同步操作的统一顺序。
- $W$ ——看到的写函数，对于  $A$  中每个读  $r$ ，它给出  $W(r)$  ( $E$  中  $r$  看到的写操作)。
- $V$ ——写入的值的函数，对于  $A$  中每个写，给出  $V(w)$  ( $E$  中  $w$  写下的值)。
- $sw$ ——与...同步，同步操作之上的偏序。
- $hb$ ——发生之前，操作之上的偏序。

注意，“与...同步”和之前发生是由执行的其他元素和格式良好的规则惟一确定的 (17.4.7 节)。

如果执行的操作组是之前发生一致的，那么该执行就是之前发生一致的 (17.4.5 节)。

### 17.4.7 格式良好的执行

我们只考虑格式良好的执行。如果下面的条件是真的，那么执行  $E = \langle P, A, po, so, W, V, sw, hb \rangle$  就是格式良好的：

(1) 在执行中，每个读看到写到相同的变量。不稳定变量的所有读和写都是不稳定的操作。对于  $A$  中的所有读  $r$ ，我们有  $A$  中的  $W(r)$  和  $W(r).v = r.v$ 。当且仅当  $r$  是不稳定读时，变量  $r.v$  才是不稳定的，并且当且仅当  $w$  是不稳定写时，变量  $w.v$  才是不稳定的。

(2) 之前发生顺序是偏序。之前发生顺序是通过“与...同步”边和程序顺序的传递闭包给出的。它必须是有效的偏序：自反性、传递性和反对称性。

(3) 执行遵守线程内一致性。对于每个线程  $t$ ， $A$  中的  $t$  执行的操作与隔离中的程序顺序的该线程生成的操作一样，并且每个写  $w$  写下值  $V(w)$ ，假定每个读看到值  $V(W(r))$ 。每个读看到的值是由内存模型决定的。给出的程序顺序必须反映操作将根据  $P$  的线程内语义执行的程序顺序。

(4) 执行是之前发生一致的 (17.4.6 节)。

(5) 执行遵守同步顺序一致性。对于  $A$  中的所有不稳定读  $r$ ，不是这样的情形： $so(r, W(r))$  或在  $A$  中存在着一个写，使得  $w.v = r.v$  和  $so(W(r), w)$  和  $so(w, r)$ 。

### 17.4.8 执行和因果关系需求

格式良好的执行  $E = \langle P, A, po, so, W, V, sw, hb \rangle$  是通过提交来自  $A$  的操作进行验证的。如果  $A$  中的所有操作可以被提交, 那么执行就满足 Java 编程语言内存模型的因果关系需求。

从作为  $C_0$  的空集合开始, 我们执行一连串的步骤, 其中我们从操作集合  $A$  采取一些操作, 并将它们添加到一组提交的操作  $C_i$ , 以获取一组新的提交的操作  $C_{i+1}$ 。要证明这是合理的, 对于每个  $C_i$ , 我们需要证明执行  $E_i$  包含满足某些条件的  $C_i$ 。

正式地, 当且仅当存在下面条件时, 执行  $E$  满足才满足 Java 编程语言内存模型的因果关系需求:

- 操作集合  $C_0, C_1, \dots$  使得:

- ◆  $C_0$  是空集合
- ◆  $C_i$  是  $C_{i+1}$  的恰当子集
- ◆  $A = \cup (C_0, C_1, C_2, \dots)$

如果  $A$  是有限的, 那么序列  $C_0, C_1, \dots$  将是有限的, 在集合  $C_n = A$  中结束。但如果  $A$  是无限的, 那么列  $C_0, C_1, \dots$  可能是无限的, 并且必须是这样的情形: 此无限序列的所有元素的联合等于  $A$ 。

- 格式良好的执行  $E_1, \dots$ , 其中  $E_i = \langle P, A_i, po_i, so_i, W_i, V_i, sw_i, hb, O_i \rangle$ 。

给定这些操作  $C_0, \dots$  和执行  $E_1, \dots$  集合, 那么  $C_i$  中的每个操作必须是  $E_i$  中的操作之一。 $C_i$  中所有操作必须共享  $E_i$  和  $E$  中相同的之前发生顺序和同步顺序。

- (1)  $C_i$  是  $A_i$  的子集

- (2)  $hb_i|C_i = hb|C_i$

- (3)  $so_i|C_i = so|C_i$

在  $E_i$  和  $E$  中, 由  $C_i$  中的写所写出的值必须相同。只有  $C_{i+1}$  中的读需要看到  $E_i$  中的写和  $E$  中的写是相同的。正式地,

- (4)  $V_i|C_i = V|C_i$

- (5)  $W_i|C_{i+1} = W|C_{i+1}$

$E_i$  中不在  $C_{i+1}$  中的所有读必须看到它们之前发生的写。 $C_i - C_{i+1}$  中的每个读  $r$  必须看到在  $E_i$  和  $E$  中看到  $C$  中的写, 但可能  $E$  中的写不同于它在  $E_i$  中看到的写。正式地,

- (6) 对于  $A_i - C_{i+1}$  中的任何读  $r$ , 我们有  $hb_i(W_i(r), r)$ 。

- (7) 对于  $(C_i - C_{i+1})$  中的任何读  $r$ , 我们在  $C_{i+1}$  中有  $W_i(r)$ , 在  $C_{i+1}$  中有  $W(r)$ 。

给定  $E_i$  一组足够的“与……同步”边, 如果有一个正在提交的操作之前发生 (17.4.5 节) 的释放获取对, 那么该对必须在所有  $E_j$  中存在, 其中  $j \geq i$ 。正式地,

- (8) 令  $ssw_i$  为  $sw_i$  边, 这些边也在  $hb_i$  的传递归纳中, 但不在  $po$  中。我们称  $ssw_i$  为与  $E_i$  的边足够地相同步。如果  $ssw_i(x, y)$  和  $hb_i(y, z)$  及  $z$  在  $C_i$  中, 那么对于所有  $j \geq i$ ,  $sw_j(x, y)$ 。

如果操作  $y$  被提交了, 那么  $y$  之前发生的所有外部操作也提交了。

- (9) 如果  $y$  在  $C_i$  中,  $x$  是外部操作和  $hb_i(x, y)$ , 那么  $x$  就在  $C_i$  中。

## 讨论

之前发生一致性是必需的，但不是足够的约束集合。只强制之前发生一致性将允许不可接受的行为——违反我们为程序建立的需求。例如，之前发生一致性允许值出现“无中生有”。在跟踪 17.6 的详细示例中，我们可以看到这一点。

跟踪 17.6 之前发生一致性不是足够的

线程 1	线程 2
<code>r1=x;</code> <code>if(r1 != 0) y = 1;</code>	<code>r2=y;</code> <code>if(r2 != 0) x=1;</code>

跟踪 17.6 中展示的代码正确地进行了同步。这可能看起来是奇怪的，因为它没有执行任何同步操作。但请记住，程序是正确进行同步的，前提是当它以连接一致方式执行时，没有数据争用。如果此代码以连接一致方式执行，那么每个操作将以程序顺序发生，两个写都不会发生。由于没有写发生，所以就不可能有数据争用：程序是正确地进行同步的。

由于此程序正确进行了同步，所以我们可以允许的仅有一些行为是连续一致行为。但由此程序的执行，该程序是之前发生一致的，而不是连续一致的：

```
r1=x; //see write of x=1
y=1
r2=y; //sees write of y=1
x=1;
```

此结果是之前发生一致的：没有防止它发生的之前发生关系。但它显然不是可接受的：没有会导致此行为的连续一致执行。我们允许读以查看执行顺序中后面到来的写的事实有时可导致不可接受的行为。我们允许读来查看执行顺序中的后面到来的写的事实有时可能导致不可接受的行为。

尽管允许读来查看执行顺序中后面到来的写有时是不需要的，但它有时也是需要的。如我们在上面看到，跟踪 17.5 需要一些读以查看在执行顺序中后面发生的写。由于读在每个线程中首先到来，所以执行顺序中恰好第一个操作必须是一个读。如果读不能看到后面发生的写，那么它就不能看到任何值，除了它读取的变量的初始值。这显然没有反映所有行为。

我们称当读可以看到未来写的问题为因果关系，因为在像跟踪 17.6 中找到的一种情形的情形会产生问题。在该情形中，读导致写发生，并且写导致读发生。对于这些操作，没有“首先导致”。因此，内存模型需要一致的方式来确定哪些读可以早点看到写。

如跟踪 17.6 看到的一个示例展示了当声明读是否可以看到在执行中后面发生的写时，规范必须是仔细的（谨记，如果读看到了在执行中后面发生的写，它就表明这样的事实：写实际是早期执行的）。

内存模型作为输入接受：一个给定的执行和一个程序，并确定执行是否是程序的合法执行。它通过下述方式来执行这项工作：逐步构建一组“已提交”的操作，这些操作反应了哪些操作被程序执行。通常，要提交的下一个操作将反映可以被连续一致执行的下一个操作。但要反映需要以后看到写的读，我们允许某些操作比它们之前发生的其他操作更早提交。

显然，一些操作可能早点提交，一些操作可能不是。例如，如果跟踪 17.6 中的写之一在那个变量之前提交，那么读就可以看到写。并且“无中生有”结果可能发生。非正式地，如果我们知道在没有假定一些数据争用发生的情况下操作可能发生，那我们就允许操作早点提交。在跟踪 17.6 中，我们不能早期执行地任何一个写，因为写不能发生，除非读看到数据争用的结果。

#### 17.4.9 可观察行为和非终止执行

对于在某些始终在某个有限时间周期内终止的程序来说，它们的行为可以按照它们可允许的（非正式地）简单地进行理解。对于可能在限定的时间量内终止的那些程序，更多微妙的问题产生了。

一个程序的可观察行为是通过程序可以执行的有限外部操作集合定义的。例如，只打印“Hello”的程序永远只是通过一组行为描述，对于任何非负整数  $i$ ，这一组行为包含了打印  $i$  次“Hello”的行为。

终止没有显式建模为行为，但可以容易地对程序进行扩展，以便生成所有的线程终止时发生的其他外部操作 *executionTermination*。

我们也定义特殊的挂起操作。如果行为是由一组包含挂起操作的外部操作描述，那么它就在外部操作被观察的地方指出一个行为，程序可以在不执行任何其他外部操作或终止的情况下运行不限数量的次数。如果所有的线程被阻塞，或者如果程序可以没有在没有执行任何外部操作的情况下执行不限数量的操作，那么程序就可以挂起。

在各种情况下，线程可能阻塞，比如当它尝试获得一个锁或执行一个依赖于外部数据的外部操作（比如读）。如果线程处在这种状态，`Thread.getState` 将返回 `BLOCKED` 或 `WAITING`。

一个执行可以导致线程被不确定地阻塞，并且执行不终止。在这种情况下，被阻塞的线程生成的操作必须包含那个线程生成的所有操作，一直到包含导致线程被阻塞的操作，并且那个操作后线程不会生成任何操作。

要推究可观察行为的原因，我们需要讨论几组可观察行为。

如果  $O$  是执行  $E$  的一组可观察行为，那么集合  $O$  必须是  $E$  的操作的一个子集，并且必须只包含有限数量的操作，即使  $A$  包含不限数量的操作。而且，如果操作  $y$  在  $O$  中，并且  $hb(x, y)$  或  $so(x, y)$ ，那么  $x$  在  $O$  中。

注意，一组可观察的操作没有被限制到外部操作。相反，只有在一组可观察操作中的外部操作被认为是可观察的外部操作。

当且仅当  $B$  是一组有限的外部操作并且是下面两种行为之一，行为  $B$  就是程序  $P$  的可允许行为：

- 存在着  $P$  的执行  $E$  和一组  $E$  的可观察操作  $O$ ，并且  $B$  是  $O$  中的外部操作集合（如果  $E$  中的任何线程在阻塞状态结束，并且  $O$  包含  $E$  中的所有操作，那么  $B$  也可能包含挂起操作），或
- 存在一组操作  $O$ ，使得  $B$  包含一个挂起操作，加上  $O$  中所有外部操作，并且对于所



有  $K \geq |O|$ , 存在着具有操作  $A$  的  $P$  的一个执行  $E$ , 并且存在着一组操作  $O'$ , 使得:

- ◆  $O$  和  $O'$  是实现可观察操作的集合的需要的  $A$  的子集。
- ◆  $O \subseteq O' \subseteq A$ 。
- ◆  $|O'| \geq K$ 。
- ◆  $O' - O$  没有包含外部操作。

### 讨论

注意, 行为  $B$  没有描述  $B$  中外部操作被观察的顺序, 但有关外部操作如何生成和执行的其他 (内部) 约束可以强加这些约束。

## 17.5 Final 字段语义

字段声明 `final` 是一次初始化的, 但在一般情况下是不改变的。`final` 字段的详细语义有点不同于普通字段的那些语义。尤其编译器有大量的自由来跨同步障碍移动 `final` 字段的读及对任意或未知方法的调用。相应地, 编译器被允许保存缓存在寄存器中的 `final` 字段, 并且在非 `final` 字段必须重新加载的情况下没有从内存中重新加载它。

`final` 字段也允许程序员在不同步的情况下实现线程安全不可变对象。线程安全不可变对象被所有的线程看成不可变的, 即使数据争用用于在线程之间将引用传递给不可变对象。对于通过不正确或恶意的代码误用不可变类, 这可以提供安全保证。`final` 字段必须正确使用, 以便提供不可变的安全保证。

当对象的构造函数完成时, 对象被认为是完全初始化的。只可以看到那个对象 (已经被完全初始化) 后面的对象的引用的线程保证可以看到那个对象的 `final` 字段的正确初始化值。

`final` 字段的使用模型是一个简单的模型。在那个对象的构造函数中设置对象的 `final` 字段。不要写出下面的位置构造的对象的引用: 在对象的构造函数完成前另一个线程可以看到它。如果情况是这样, 那么当对象被另一个线程看到时, 那个线程将始终看到该对象的 `final` 字段的正确构造版本。它也将看到与 `final` 字段一样新的 `final` 字段引用的的任何对象或数组。

### 讨论

下面的示例展示了 `final` 字段与普通字段的比较。

```
class FinalFieldExample {
    final int x;
    int y;
    static FinalFieldExample f;
    public FinalFieldExample() {
        x = 3;
        y = 4;
    }
}
```



```
static void writer() {
    f = new FinalFieldExample();
}
static void reader() {
    if (f != null) {
        int i = f.x; // guaranteed to see 3
        int j = f.y; // could see 0
    }
}
```

类 `FinalFieldExample` 有一个最终 `int` 字段 `x` 和非最终 `int` 字段 `y`。一个线程可以执行方法 `writer()`，并且另一个线程可以执行方法 `reader()`。

因为在对象的构造函数完成后，`writer()` 写出了 `f`，所以 `reader()` 将被保证看到 `f.x` 的正确初始化值：它将读取值 3。但 `f.y` 不是最终的：因此 `reader()` 方法不被保证看到它的值 4。

### 讨论

`final` 字段被设计成允许必要的安全保证。考虑下面的例子，一个线程（我们将称为线程 1）执行：

```
Global.s = "/tmp/usr".substring(4);
```

而另一个线程（线程 2）执行：

```
String myS = Global.s;
If (myS.equals("/tmp*)) System.out.println(myS);
```

`String` 对象打算成为不可变的，并且字符串操作没有执行同步。尽管 `String` 实现没有任何数据争用，但其他代码可能有涉及字符串使用的数据争用，并且内存模型对有数据争用的程序做出微弱的保证。特别地，如果 `Strings` 类的字段不是最终的，那么下面的情况将是可能的（即使不像）：线程 2 最终可以看到字符串对象的偏移的默认值 0，并允许它与 `"/tmp"` 视为同等。`String` 对象后一个操作可以看到 4 的正确偏移，以便 `String` 被感知为 `"/usr"`。Java 程序语言的许多安全特性依赖于感知为真正不可变的 `Strings`，即使恶意代码正在使用数据争用在线程之间传递 `String` 引用。

## 17.5.1 Final 字段的语义

`final` 字段的语义如下所示。令 `o` 是一个对象，`c` 是在其中编写 `f` 的构造函数。当 `c` 存在时，`o` 的 `final` 字段上的冻结操作的正常或突然发生。

### 讨论

注意，如果一个构造函数调用另一个构造函数，并且被调用的构造函数设置了 `final` 字段，那么 `final` 字段的冻结在被调用的构造函数结束时发生。

对于每个执行,读的行为被其他的两个偏序影响,即解引用链 *dereferences()* 和内存链 *mc()*——它们被认为是执行的一部分(从而对于特定的执行是固定的)。这些偏序必须满足下面的约束(这些约束需要有惟一的解决方案):

- **解引用链:** 如果操作 *a* 通过没有初始化对象 *o* 的一个线程 *t* 对该对象的字段或元素的读或写,那么必须有通过线程 *t* 的某个读 *r*,以便看到 *o* 的地址,使得 *r dereferences(r, a)*。
- **内存链:** 在内存链接顺序上有几个约束:
  - ◆ 如果 *r* 是一个看到写 *w* 的读,那么它必须是 *mc(w, r)* 的情形。
  - ◆ 如果 *r* 和 *a* 是使得 *dereferences(r, a)* 的操作,那么它必须是 *mc(r, a)* 的情形。
  - ◆ 如果 *w* 是通过没有初始化对象 *o* 的线程 *t* 对对象 *o* 的地址的写,那么必须存在通过线程 *t* 的某个读 *r*,以便查看到使得 *mc(r, w)* 的 *o* 的地址。

给定一个写 *w*、一个冻结 *f*、操作 *a* (不是 final 字段的读)、一个通过 *f* 冻结的 final 字段的读 *r<sub>1</sub>* 和一个读 *r<sub>2</sub>*,使得 *hb(w, f)*, *hb(f, a)*, *mc(a, r<sub>1</sub>)* 和 *dereferences(r<sub>1</sub>, r<sub>2</sub>)*,那么当确定那些值可以被 *r<sub>2</sub>* 看到时,我们考虑 *hb(w, r<sub>2</sub>)* (但这些顺序没有用其他之前发生顺序进行传递关闭)。注意,解引用顺序是反射性的,并且 *r<sub>1</sub>* 可与 *r<sub>2</sub>* 相同。

对于 final 字段的读,被认为要在 final 字段读之前到来的仅有一些写是通过 final 字段语义推导的写。

### 17.5.2 在构造期间阅读 Final 字段

构造对象的线程中的该对象的 final 字段的读的排序,是与通过通常的之前发生规则对构造函数中的该字段进行的初始化有关的。如果在构造函数中设置字段后读发生了,那么它就会看到被分配的 final 字段的值,否则它看到默认值。

### 17.5.3 Final 字段的后续修改

在一些情形下,比如反序列化,系统将需要在构造后更改对象的 final 字段。final 字段可以通过反射和其他实现独立方式进行改变。此具有合理语义的惟一模式是这种模式:对象被构造,然后对象的 final 字段被更改。对象不应该对于其他线程可见,final 字段也不应该被读取,直到对象的 final 字段的所有更新已完成。final 字段的冻结发生在 final 字段被设置的构造函数结束,及通过反射或其他特殊机制对 final 字段的每个修改之后。

尽管那样,也有大量的应用程序。如果在字段声明中将字段初始化为编译时常量,那么对 final 字段的更改可能不会被观察到,因为使用此 final 字段是在编译时用编译时常量进行替换的。

另一个问题是规范允许对字段的强制优化。在线程中,借助没有在构造函数是发生的 final 字段的修改来对 final 字段的读进行重新排序是允许的。

## 讨论

例如，考虑下面的代码段：

```
class A {
    final int x;
    A() {
        x = 1;
    }
    int f() {
        return d(this, this);
    }
    int d(A a1, A a2) {
        int i = a1.x;
        g(a1);
        int j = a2.x;
        return j - i;
    }
    static void g(A a) {
        // uses reflection to change a.x to 2
    }
}
```

在 `d()` 方法中，编译器被允许对 `x` 的读进行重排序，并且自由调用 `g()`。因此，`A().f()` 可能返回 -1、0 或 1。

实现可以提供一种方式，以便在 `final` 字段上下文中执行代码块。如果对象是在 `final` 字段安全上下文中构造的，那么对象的 `final` 字段的读将不会借助于此 `final` 字段在该最终字段安全上下文中发生的修改进行重新排序。

`final` 字段上下文有其他的操作。如果线程已经看到对象的一个不正确的发布引用，并且该对象允许线程看到 `final` 字段的默认值，然后，在 `final` 字段安全上下文中，读取对象的正确发布引用，那么可以保证它能够看到 `final` 字段的正确值。从形式上讲，在 `final` 字段安全上下文中执行的代码被认为是一个不同的线程（只针对 `final` 字段语义的用途）。

在实现中，编译器不应该将对 `final` 字段的访问移入或移出 `final` 字段安全上下文（尽管它可以在这样的上下文的执行周围移动，只要对象没有在那个上下文中进行构造）。

使用 `final` 字段安全上下文恰当的一个地方是执行器或线程池。通过在独立的 `final` 字段安全上下文中执行每个 `Runnable`，执行器可以保证一个 `Runnable` 对对象 `o` 的不正确访问不会删除相同执行器处理的其他 `Runnable` 的 `final` 字段保证。

#### 17.5.4 写保护字段

通常，最终静态字段不可以被修改。但 `System.in`、`System.out` 和 `System.err` 是这样的一些最终静态字段：出于遗留原因，这些字段必须被允许通过方法 `System.setIn`、`System.setOut` 和 `System.setErr` 来进行改变。我们将这些字段称为被写保护以将它们从普通的 `final` 字段区分开来。

编译器需要以与其他 `final` 字段不同的方式处理这些字段。例如，普通 `final` 字段的读是对同步的“免疫”：在锁或不稳定读中涉及的障碍不会影响从 `final` 字段读取什么值。由于写保护字段的值可以被视为更改，所以同步事件应该对它们有影响。因此，语义规定了这些字段可以被当成普通字段，这些普通字段不能可以由用户代码进行更改，除非用户代码在 `System` 类。

## 17.6 字分开

Java 虚拟机的一种实现考虑是每个字段和数组元素被认为是不同的：对一个字段或元素的更新不须与任何其他字段或元素的读或更新交互。独立更新字节数组的邻接元素的两个线程尤其不需要干预或交互，并且不需要同步来确保连续一致性。

一些处理器没有提供能力写到单个字节。通过下述方式在这样的一个处理器上实现字节数组更新会是非法的：简单地读取整个字，更新恰当的字节，然后将整个字写回到内存。这种问题有时称为字分开，并且在不能容易更新隔离中的单个字节的处理器上，将需要某种其他方法。

下面是检测字分开的测试案例：

```
public class WordTearing extends Thread {
    static final int LENGTH = 8;
    static final int ITTERS = 1000000;
    static byte[] counts = new byte[LENGTH];
    static Thread[] threads = new Thread[LENGTH];
    final int id;
    WordTearing(int i) {
        id = i;
    }
    public void run() {
        byte v = 0;
        for (int i = 0; i < ITTERS; i++) {
            byte v2 = counts[id];
            if (v != v2) {
                System.err.println("Word-Tearing found: " +
                    "counts[" + id
                        + "] = " + v2 + ", should be " + v);
                return;
            }
            v++;
            counts[id] = v;
        }
    }
}
```

```
public static void main(String[] args) {  
    for (int i = 0; i < LENGTH; ++i)  
        (threads[i] = new WordTearing(i)).start();  
}
```

这产生了必须将字节重写到邻接节点的点。

## 17.7 double 和 long 的非原子处理

一些实现可能发现，将 64 位长或 double 值上的单个写操作分成邻接 32 位值上的两写操作会方便些。出于效率的考虑，这种行为是特定于实现的；Java 虚拟机能够以自由的方式原子地或分成两部分来写到 long 和 double 值。

出于 Java 程序语言模型的目的，一个稳定 long 或 double 值的写被当成两个不同的写处理：一个针对各半的 32 位。这可能导致这样的情形：线程从一个写中看到 64 位的前 32 位，并且从另一个写看到第二个 32 位。不稳定的 long 和 double 值的写和读始终是原子的。引用的写和读始终是原子的，而不管它们是否实现为 32 或 64 位值。

VM 实现者被鼓励尽可能分开 64 位。程序员被鼓励声明共享的 64 位值声明为非稳定的或正确地同步他们的程序，以避免可能的并发问题。

## 17.8 等待集合和通知

### 17.8.1 等待

等待操作在调用 wait() 或时间形式 wait(long millisecs) 和 wait(long millisecs, int nanosecs) 之后发生。具有参数 0 的 wait(long millisecs) 的调用，或具有两个 0 参数的 wait(long millisecs, int nanosecs) 的调用，等价于对 wait() 的调用。

如果线程在没有抛出 InterruptedException 的情况下返回，那么线程就从 wait 正常返回。

令线程  $t$  是在对象  $m$  上执行等待方法的线程，并令  $n$  是  $t$  在  $m$  上加锁操作的编号，这些操作已经不被解锁操作匹配。下面的操作之一发生。

- 如果  $n$  是 0 (也就是，线程  $t$  已经没有占用目标  $m$  的锁)，则抛出 IllegalMonitorStateException。
- 如果这是一个定时等待，并且十亿分之一秒参数不在 0~999999，或者毫秒参数是负的，那么就抛出 IllegalArgumentException。
- 如果线程  $t$  被中断，那就抛出 InterruptedException，并将  $t$  的中断状态设置为假。



• 否则，下面的序列发生：

(1) 线程  $t$  被添加到对象  $m$  的等待组中，并在  $m$  上执行  $n$  个解锁操作。

(2) 线程  $t$  没有执行任何的进一步指令，直到它已经从  $m$  的等待集合中删除。由于下面的操作的任何之一，该程序可能从等待集合中删除，并在后面的某个时间继续。

- ◆ 在从等待集合中删除而选择的  $t$  的  $m$  上正被执行的 `notify` 操作。
- ◆ 在  $m$  上正被执行的 `notifyAll` 操作。
- ◆ 在  $t$  上执行的 `interrupt` 操作。
- ◆ 如果这是一个定时等待，则为  $m$  的等待集合删除的内部操作，该集合至少在 `millisecs` 毫秒加 `nanosecs` 十亿分之一秒消逝后发生（从写操作开始）。
- ◆ 根据实现的内部操作。实现被允许（尽管不鼓励）执行“伪造的唤醒”——以便从等待集合中删除线程，从而能够在没有显式指令这样做的情况下再继续。注意这种装备成了在循环中使用 `wait` 的 Java 编码实践的必要条件，这些循环只有在线程等待持有的某个逻辑条件时才终止。

每个线程必须通过可能导致它从等待集合中删除的事件确定顺序。顺序不一定与其他排序一致，但线程表现得像以那个顺序发生的那些事件一样。

例如，如果线程  $t$  在  $m$  的等待队列中，然后  $t$  的中断和  $m$  的通知发生，那么在这些事件上必须有一个顺序。如果中断被认为首先发生，那么  $t$  将最终通过抛出 `InterruptedException` 来从 `wait` 中返回，并且  $m$  中的等待集合中的某个其他线程（如果在通知的时候存在的话）必须接收通知。如果通知被认为是首先发生的，那么  $t$  将最终正常地从 `wait` 返回，且中断仍然挂起。

(3) 线程  $t$  在  $m$  上执行  $n$  个加锁操作。

(4) 如果由于中断线程  $t$  在步骤 2 中从  $m$  的等待集合中删除了，那么  $t$  的中断状态就被设置为假，并且等待方法抛出 `InterruptedException`。

## 17.8.2 通知

通知操作在调用方法 `notify` 和 `notifyAll` 调用之后发生。令线程  $t$  是执行对象  $m$  上的这些方法的任一方法的线程，并令  $n$  是  $t$  在  $m$  上的加锁操作的数量，这些操作没有被解锁操作匹配。下面操作之一发生了。

- 如果  $n$  是 0，则抛出 `IllegalMonitorStateException`。情形是这样的：线程  $t$  已经没有占有目标  $m$  的锁。
- 如果  $n$  大于 0，并且这是一个 `notify` 操作，那么如果  $m$  的等待集合不是空的，则是  $m$  的当前等待集合的一个成员的线程  $u$  被选择，并从等待集合中删除（不保证在等待集合中选定哪个线程）。从等待集合中进行该删除让  $u$  在等待操作中得以继续。但注意，继续之后的  $u$  的加锁操作不能成功，直到  $t$  完全解锁  $m$  的监视器后的某个时间。
- 如果  $n$  大于 0，并且这是一个 `notifyAll` 操作，那么所有的线程就从  $m$  的等待集合中删除并继续。但请注意，它们当中仅有的一个将一次锁住 `wait` 的继续期间需要



的监视器。

### 17.8.3 中断

中断操作在调用方法 `Thread.interrupt` 及定义来依次调用它的方法（比如 `ThreadGroup.interrupt`）之后发生。对于某个线程  $u$ ，令  $t$  是调用 `u.interrupt` 的线程，其中  $t$  和  $u$  可能是相同的。此操作导致  $u$  的中断状态被设置到真。

另外，如果存在着其等待集合包含  $u$  的某个对象  $m$ ，那么  $u$  就从  $m$  的等待集合中删除。这使得  $u$  能够在等待操作中继续，在该操作的情况中，此等待将在重新加锁  $m$  的监视器后抛出 `InterruptedException`。

调用 `Thread.isInterrupted` 可以确定线程的中断状态。静态方法 `Thread.interrupted` 可由线程调用来观察和清除自己的中断状态。

### 17.8.4 等待、通知和中断的交互

上面的规范允许我们确定与等待、通知和中断有关的几个属性。如果在等待时，线程同时是通知和中断的，它就可能是下面之一：

- 从 `wait` 正常返回，尽管仍然有挂起中断（在其他工作中，对 `Thread.interrupted` 的调用将返回真）。
- 通过抛出 `InterruptedException` 从 `wait` 返回。

线程不可以重置它的中断状态，并从 `wait` 的调用中正常返回。

同样，通知不能由于中断而丢失。假定线程的集合  $s$  在对象  $m$  的等待集合中，并且另一个线程在  $m$  上执行 `notify`。那么有下面之一发生：

- 至少  $s$  中有一个线程从 `wait` 中正常返回，或者
- $s$  中的所有线程必须通过抛出 `InterruptedException` 退出 `wait`。

注意，如果线程通过 `notify` 被中断和唤醒，并且线程通过抛出 `InterruptedException` 从 `wait` 返回，那么等待集合中的某个其他线程必须被通知到。

## 17.9 休眠和转交

`Thread.sleep` 导致当前执行的线程休眠（临时停止执行）指定的时间段，服从系统计时器和日程安排程序的精度和精确性。此线程没有失去任何监视器的所有权，并且执行的继续将取决于调度和可执行程序的处理器的可用性。

时间周期的休眠和转交操作都不需要有可观察的效果。

记住下面这些是重要的：`Thread.sleep` 和 `yield` 都没有任何同步语义。编译器尤其不需要在调用 `Thread.sleep` 或 `Thread.yield` 之前将缓存在寄存器上的写刷新到共享缓存中，在调用 `Thread.sleep` 或 `Thread.yield` 后，编译器也不重新加载缓存在寄存器中的值。

**讨论**

例如，在接下来的代码段中，假定 `this.done` 是不稳定的布尔字段：

```
while (!this.done)
    Thread.sleep(1000);
```

编译器仅可以自由地读取字段 `this.done` 一次，并在循环的每个执行中重用缓存值。这意味着循环将不会结束，即使另一个线程更改了 `this.done` 的值。

问：标题中存在语法吗？

答：存在。谢谢！

——Gertrude Stein, 摘自《How to Write》中“Arthur a Grammar”(1931)

本章介绍了 Java 编程语言的语法。

前面的章节中详细分开介绍的语法对于说明是很好的，但它不适合作为解析器的基础。本章中介绍的语法是参考实现的基础。注意，这不是 LL(1)语法，尽管在许多场合，我们不需要对未来进行过多的考虑。

下面的语法使用以下 BNF 样式的约定：

- $[x]$  表示出现 0 个或 1 个  $x$ 。
- $/x/$  表示出现 0 个或多个  $x$ 。
- $x|y$  意味着  $x$  或  $y$  中之一。

### 18.1 Java 编程语言的语法

*Identifier:*

*IDENTIFIER*

*QualifiedIdentifier:*

*Identifier[ . Identifier ]*

*Literal:*

*IntegerLiteral*

*FloatingPointLiteral*

*CharacterLiteral*

*StringLiteral*

*BooleanLiteral*

*NullLiteral*

*Expression:*

*Expression1* [*AssignmentOperator Expression1*]

*AssignmentOperator:*

=

+ =

- =

\* =

/ =

& =

| =

^ =

% =

<< =

>> =

>>> =

*Type:*

*Identifier* [*TypeArguments*]( . *Identifier* [*TypeArguments*]) ( [ ] )

*BasicType*

*TypeArguments:*

<*TypeArgument* / , *TypeArgument*>

*TypeArgument:*

*Type* -

? [(*extends* | *super*)*Type*]

*StatementExpression:*

*Expression*

*ConstantExpression:*

*Expression*

*Expression1:*

*Expression2* [*Expression1Rest*]

*Expression1Rest:*

? *Expression* : *Expression1*

*Expression2:*

*Expression3* [*Expression2Rest*]

*Expression2Rest:*

{*InfixOp Expression3*}

*Expression3* instanceof *Type*

*InfixOp*:

||  
 &&  
 |  
 ^  
 &  
 ==  
 !=  
 <  
 >  
 <=  
 >=  
 <<  
 >>  
 >>>  
 +  
 -  
 \*  
 /  
 %

*Expression3*:

*PrefixOp* *Expression3*  
 ( *Expression* | *Type* ) *Expression3*  
*Primary* [*Selector*] [*PostfixOp*]

*Primary*:

*ParExpression*  
*NonWildcardTypeArguments*(*ExplicitGenericInvocationSuffix* | *this Arguments*)  
*this* [*Arguments*]  
*super* *SuperSuffix*  
*Literal*  
*new Creator*  
*Identifier* [ . *Identifier* ] [ *IdentifierSuffix* ]  
*BasicType* [ ( [ ] ) ].class  
void.class

*IdentifierSuffix*:

[ ( [ ] / [ ] ) ].class | *Expression* ] )

*Arguments*

*. (class | ExplicitGenericInvocation | this | super Arguments | new  
[Non WildcardTypeArguments] InnerCreator)*

*ExplicitGenericInvocation:*

*Non WildcardTypeArguments ExplicitGenericInvocationSuffix*

*Non WildcardTypeArguments:*

*<TypeList>*

*ExplicitGenericInvocationSuffix:*

*super SuperSuffix*

*Identifier Arguments*

*PrefixOp:*

*++*

*--*

*!*

*~*

*+*

*-*

*PostfixOp:*

*++*

*--*

*Selector:Selector:*

*. Identifier [Arguments]*

*. ExplicitGenericInvocation*

*. this*

*. super SuperSuffix*

*. new [Non WildcardTypeArguments] InnerCreator*

*[ Expression]*

*SuperSuffix:*

*Arguments*

*. Identifier [ Arguments]*

*BasicType:*

*byte*

*short*



char  
int  
long  
float  
double  
boolean

*Arguments:*

( [Expression ( , Expression)] )

*Creator:*

[NonWildcardTypeArguments] CreatedName ( ArrayCreatorRest | ClassCreatorRest )

*CreatedName:*

Identifier [Non WildcardTypeArguments] [.Identifier [NonWildcardTypeArguments]]

*InnerCreator:*

Identifier ClassCreatorRest

*ArrayCreatorRest:*

( ( ) [ [ ] ] ArrayInitializer | Expression ) ( [ Expression ] ) ( [ ] ) )

*ClassCreatorRest:*

Arguments [ClassBody]

*ArrayInitializer:*

{ [VariableInitializer [,VariableInitializer] [,]] }

*VariableInitializer:*

ArrayInitializer  
Expression

*ParExpression:*

( Expression )

*Block:*

{ BlockStatements }

*BlockStatements:*

{ BlockStatement }

*BlockStatement:*

LocalVariableDeclarationStatement  
ClassOrInterfaceDeclaration

*[Identifier:]Statement*

*LocalVariableDeclarationStatement:*

*[final]Type VariableDeclarators ;*

*Statement:*

*Block*

*assert Expression [ : Expression];*

*if ParExpression Statement [else Statement]*

*for (ForControl) Statement*

*while ParExpression Statement*

*do Statement while ParExpression ;*

*try Block ( Catches | [Catches] finally Block)*

*switch ParExpression { SwitchBlockStatementGroups }*

*synchronized ParExpression Block*

*return [Expression];*

*throw Expression ;*

*break [Identifier]*

*continue [Identifier]*

*;*

*StatementExpression ;*

*Identifier : Statement*

*Catches:*

*CatchClause{CatchClause}*

*CatchClause:*

*catch ( FormalParameter ) Block*

*SwitchBlockStatementGroups:*

*{ SwitchBlockStatementGroup }*

*SwitchBlockStatementGroup:*

*SwitchLabel BlockStatements*

*SwitchLabel:*

*case ConstantExpression :*

*case EnumConstantName :*

*default :*

*MoreStatementExpressions:*

*{ , StatementExpression }*

*ForControl:*

*ForVarControl*  
*ForInit*; [*Expression*] ; [*ForUpdate*]

*ForVarControl*

[*final*][*Annotations*]*Type Identifier ForVarControlRest*

*Annotations:*

*Annotation* [*Annotations*]

*Annotation:*

@ *TypeName*{ ([*Identifier*=]*ElementValue*) }

*ElementValue:*

*ConditionalExpression*  
*Annotation*  
*ElementValueArrayInitializer*

*ConditionalExpression:*

*Expression2 Expression1Rest*

*ElementValueArrayInitializer:*

{ [*ElementValues*] [, ] }

*ElementValues:*

*ElementValue*[*ElementValues*]

*ForVarControlRest:*

*VariableDeclaratorsRest*; [*Expression*] ; [*ForUpdate*]  
: *Expression*

*ForInit:*

*StatementExpression Expressions*

*Modifier:*

*Annotation*  
public  
protected  
private  
static  
abstract  
final  
native

synchronized  
transient  
volatile  
strictfp

*VariableDeclarators:*

*VariableDeclarator* { , *VariableDeclarator* }

*VariableDeclaratorsRest:*

*VariableDeclaratorRest* { , *VariableDeclarator* }

*ConstantDeclaratorsRest:*

*ConstantDeclaratorRest* { , *ConstantDeclarator* }

*VariableDeclarator:*

*Identifier* *VariableDeclaratorRest*

*ConstantDeclarator:*

*Identifier* *ConstantDeclaratorRest*

*VariableDeclaratorRest:*

{ [ ] } [ = *VariableInitializer* ]

*ConstantDeclaratorRest:*

{ [ ] } = *VariableInitializer*

*VariableDeclaratorId:*

*Identifier* { [ ] }

*CompilationUnit:*

[ [Annotations] package *QualifiedIdentifier* ; ] { *ImportDeclaration* }  
{ *TypeDeclaration* }

*ImportDeclaration:*

import [static] *Identifier* { . *Identifier* } [ . \* ] ;

*TypeDeclaration:*

*ClassOrInterfaceDeclaration*  
;

*ClassOrInterfaceDeclaration:*

{ *Modifier* } ( *ClassDeclaration* \ *InterfaceDeclaration* )

*ClassDeclaration:*

*NormalClassDeclaration*

*EnumDeclaration*

*NormalClassDeclaration:*

*class Identifier [TypeParameters] [extends Type] [implements TypeList]*

*ClassBody*

*TypeParameters:*

*<TypeParameter{ , TypeParameter}>*

*TypeParameter:*

*Identifier [extends Bound]*

*Bound:*

*Type [& Type]*

*EnumDeclaration:*

*enum Identifier [implements TypeList] EnumBody*

*EnumBody:*

*{ [EnumConstants] [,] [EnumBodyDeclarations] }*

*EnumConstants:*

*EnumConstant*

*EnumConstants, EnumConstant*

*EnumConstant:*

*Annotations Identifier[Arguments][ClassBody]*

*EnumBodyDeclarations:*

*; {ClassBodyDeclaration}*

*InterfaceDeclaration:*

*NormalInterfaceDeclaration*

*AnnotationTypeDeclaration*

*NormalInterfaceDeclaration:*

*interface Identifier [TypeParameters][extends TypeList]*

*InterfaceBody*

*TypeList:*

*Type { , Type}*

*AnnotationTypeDeclaration:*

*@ interface Identifier AnnotationTypeBody*

*AnnotationTypeBody:*

*{ [AnnotationTypeElementDeclarations] }*

*AnnotationTypeElementDeclarations:*

*AnnotationTypeElementDeclaration*

*AnnotationTypeElementDeclarations AnnotationTypeElementDeclaration*

*AnnotationTypeElementDeclaration:*

*{Modifier}AnnotationTypeElementRest*

*AnnotationTypeElementRest:*

*Type Identifier AnnotationMethodOrConstantRest ;*

*ClassDeclaration*

*InterfaceDeclaration*

*EnumDeclaration*

*AnnotationTypeDeclaration*

*AnnotationMethodOrConstantRest:*

*AnnotationMethodRest*

*AnnotationConstantRest*

*AnnotationMethodRest:*

*( ){DefaultValue}*

*AnnotationConstantRest:*

*VariableDeclarators*

*DefaultValue:*

*default ElementValue*

*ClassBody:*

*{ [ClassBodyDeclaration] }*

*InterfaceBody:*

*{ [InterfaceBodyDeclaration] }*

*ClassBodyDeclaration:*

*;*

*[static]Block*

*{Modifier}MemberDecl*

*MemberDecl:*



*GenericMethodOrConstructorDecl*  
*MethodOrFieldDecl*  
*void Identifier VoidMethodDeclaratorRest*  
*Identifier ConstructorDeclaratorRest*  
*InterfaceDeclaration*  
*ClassDeclaration*

*GenericMethodOrConstructorDecl:*  
    *TypeParameters GenericMethodOrConstructorRest*

*GenericMethodOrConstructorRest:*  
    *(Type | void) Identifier MethodDeclaratorRest*  
    *Identifier ConstructorDeclaratorRest*

*MethodOrFieldDecl:*  
    *Type Identifier MethodOrFieldRest*

*MethodOrFieldRest:*  
    *VariableDeclaratorRest*  
    *MethodDeclaratorRest*

*InterfaceBodyDeclaration:*  
    ;  
    *{Modifier} InterfaceMemberDecl*

*InterfaceMemberDecl:*  
    *InterfaceMethodOrFieldDecl*  
    *InterfaceGenericMethodDecl*  
    *void Identifier VoidInterfaceMethodDeclaratorRest*  
    *InterfaceDeclaration*  
    *ClassDeclaration*

*InterfaceMethodOrFieldDecl:*  
    *Type Identifier InterfaceMethodOrFieldRest*

*InterfaceMethodOrFieldRest:*  
    *ConstantDeclaratorsRest;*  
    *InterfaceMethodDeclaratorRest*

*MethodDeclaratorRest:*  
    *FormalParameters [ { } ][throws QualifiedIdentifierList](MethodBody !; )*

*VoidMethodDeclaratorRest:*

*FormalParameters*[throws *QualifiedIdentifierList*]( *MethodBody* | ; )

*InterfaceMethodDeclaratorRest*:

*FormalParameters*[ [ ] ][throws *QualifiedIdentifierList*] ;

*InterfaceGenericMethodDecl*:

*TypeParameters*(*Type* | *void*)*Identifier* *InterfaceMethodDeclaratorRest*

*VoidInterfaceMethodDeclaratorRest*:

*FormalParameters* [throws *QualifiedIdentifierList*] ;

*ConstructorDeclaratorRest*:

*FormalParameters*[throws *QualifiedIdentifierList*]*MethodBody*

*QualifiedIdentifierList*:

*QualifiedIdentifier* { , *QualifiedIdentifier* }

*FormalParameters*:

( [ *FormalParameterDecls* ] )

*FormalParameterDecls*:

[ *final* ] [ *Annotations* ] *Type* *FormalParameterDeclsRest*

*FormalParameterDeclsRest*:

*VariableDeclaratorId* [ , *FormalParameterDecls* ]

...*VariableDeclaratorId*

*MethodBody*:

*Block*

*EnumConstantName*:

*Identifier*